FORTRAN Version 2 for NOS/VE Language Definition

CONTROL DATA

Usage

Sharon Lammers

FORTRAN Version 2 for NOS/VE
Language Definition

Usage

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features and parameters.

Publication Number 60487113

Manual History

Revision	System Version/ PSR Level	Product Level	Date
A	1.2.2/678	2.0	April 1987
B	1.2.3/688	2.1	September 1987
C	1.3.1/700	2.2	April 1988
D	1.4.1/716	2.3	December 1988

This revision reflects technical and editorial changes as well as the following new capabilities:

- FLP$FORMAT_MESSAGE subprogram
- OPTIMIZATIONS_OPTION=ALTERNATIVE_CODE_SELECTION on the VECTOR_FORTRAN command
- Preprocessor C$ directives

The Keyed-File Interface and Sort/Merge chapters have been moved from this manual to a new and separate manual titled FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces, publication number 60485917.

Contents

About This Manual 7

- Audience 7
- Organization 8
- Conventions 9
- Control Data Extensions 10
- Submitting Comments 10
- CYBER Software Support Hotline 10

Introduction to NOS/VE FORTRAN 1-1

- Standard FORTRAN Capabilities . . 1-1
- Control Data Extensions 1-2
- Input and Output of the FORTRAN Compiler 1-4
- The NOS/VE Environment 1-6

Language Elements 2-1

- The FORTRAN Character Set 2-2
- FORTRAN Statements 2-4
- Statement Order 2-7
- Symbolic Names 2-9
- Constants 2-10
- Variables 2-23

Arrays 3-1

- Declaring Arrays 3-1
- Declaring Allocatable Arrays 3-5
- Array Shape and Conformability . . 3-7
- Array Storage 3-8
- Array References 3-10

Specification Statements 4-1

- BOOLEAN Statement 4-3
- BYTE Statement 4-4
- CHARACTER Statement 4-5
- COMMON Statement 4-8
- COMPLEX Statement 4-11
- DATA Statement 4-12
- DIMENSION Statement 4-18
- DOUBLE PRECISION Statement . 4-20
- EQUIVALENCE Statement 4-21
- EXTERNAL Statement 4-24
- IMPLICIT Statement 4-26
- INTEGER Statement 4-28
- INTERFACE and END INTERFACE Statements 4-30
- INTRINSIC Statement 4-32
- LOGICAL Statement 4-34
- PARAMETER Statement 4-35
- REAL Statement 4-37
- SAVE Statement 4-38

Expressions and Assignment Statements 5-1

- Expressions 5-1
- Assignment Statements 5-25

Flow Control Statements 6-1

- GO TO Statements 6-1
- IF Statements 6-5
- WHERE Statements 6-10
- DO Statement 6-13
- CONTINUE Statement 6-21
- PAUSE Statement 6-22
- STOP Statement 6-23
- END Statement 6-24

Input/Output 7-1

- Records 7-1
- Files 7-2
- Input/Output Units 7-8
- Input/Output Statement Specifiers 7-11
- Input/Output Lists 7-13
- Formatted Input/Output 7-18
- Unformatted Input/Output 7-59
- List Directed Input/Output 7-61
- Namelist Input/Output 7-65
- Buffer Input/Output 7-71
- Mass Storage Input/Output 7-74
- Direct Access Files 7-83
- Internal Input/Output 7-86
- Segment Access Files 7-93
- Input/Output-Related Statements and Routines 7-95
- FORTRAN Fast I/O 7-119

Program Units **8-1**

Overview of Program Units and Statement Functions............. 8-2
Main Programs 8-4
Subprograms 8-7
Statement Functions 8-17
How Program Units Pass Values . 8-20

Intrinsic Functions **9-1**

Generic and Specific Names 9-3
Function Descriptions 9-8

NOS/VE and Utility Subprograms.................... **10-1**

NOS/VE Status Subprograms 10-2
Communicating with NOS/VE ... 10-11
Utility Subprograms 10-38

C$ Directives **11-1**

Listing Control 11-2
Conditional Compilation 11-4
Loader Control 11-6
Collation Control 11-8
DO Loop Control 11-9
External Control 11-10
Vectorization Control 11-11
C$ PARAM Directive 11-13
Preprocessor Control 11-15

Compilation and Execution **12-1**

VECTOR_FORTRAN Command .. 12-1
Compiler Output Listing 12-24
Execution Command 12-40

Vectorization **13-1**

What is Vectorization? 13-2
Why is Vectorization Faster? 13-4
What is Explicit Vectorization? .. 13-6
What is Implicit Vectorization? .. 13-9
How the Compiler Selects Statements for Implicit Vectorization.................. 13-11
Statements That Cannot be Vectorized Implicitly........... 13-13
Vector Operations 13-15
Controlling Implicit Vectorization with C$ Directives............. 13-29
Producing the Implicit Vectorization Report............ 13-31
Implicit Vectorization Messages . 13-35

Examples **14-1**

Program PASCAL 14-1
Program CORR 14-3
Program COMPSAL 14-6
Subroutine COUNTC 14-8
Mass Storage Input/Output Examples...................... 14-10
Program SCLCALL 14-12

Glossary **A-1**

Related Manuals **B-1**

Ordering Printed Manuals B-1
Accessing Online Manuals B-1

ASCII Character Set and Collating Weight Tables........... **C-1**

Calling Other Language Subprograms..................... **D-1**

Calling CYBIL Procedures D-1
Calling COBOL Subprograms D-5
Calling C Routines D-7

Introduction to Debug **E-1**

Using Debug E-2
How to Get Help E-4
Screen Debug Example E-5

The Programming Environment and the Professional Programming Environment **F-1**

Programming Environment F-1
Professional Programming Environment..................... F-5

Control Data Extensions to Standard FORTRAN.............. **G-1**

Language Summary H-1

Assignment H-1
Multiple Assignment H-1
Type Declaration H-2
External Declaration H-2
Intrinsic Declaration H-2
Storage Allocation H-3
Flow Control H-5
Main Program H-6
Subprogram H-6
Statement Function H-6
Subroutine Call H-6
Function Reference H-6
Entry Point H-7
Return H-7
Formatted Input/Output H-7
Unformatted Input/Output H-7
List Directed Input/Output H-7
Direct Access Input/Output H-8
Namelist Input/Output H-8
Buffer Input/Output H-9
Internal Data Transfer H-9
Format Specification H-10
File Positioning H-12
File Status H-12

Differences Between FORTRAN Version 1 and FORTRAN Version 2 .... I-1

General Differences I-1
Differences Due to Vectorization .. I-4

Figures

1-1. The FORTRAN Compiler 1-4
2-1. FORTRAN Program Example ... 2-4
3-1. Storage Pattern of a One-Dimensional Array 3-3
3-2. Storage Pattern of a Two-Dimensional Array 3-3
3-3. Storage Pattern of a Three-Dimensional Array 3-4
8-1. Main Program, Function, and Subroutine Example 8-3
13-1. The Scatter Instruction 13-16
13-2. The Gather Instruction 13-17
13-3. The Sequence Vector 13-18
13-4. Sum Reduction 13-19
13-5. Scalar Expansion 13-20
13-6. Broadcasting 13-21
13-7. Intrinsic Promotion 13-22
13-8. Stripmining 13-26
13-9. Loop Collapse 13-28
14-1. Program PASCAL 14-1
14-2. Program PASCAL Output 14-2
14-3. Program CORR 14-4
14-4. Program CORR Output 14-5
14-5. Program COMPSAL 14-6
14-6. Sample Terminal Dialog for Program COMPSAL 14-7
14-7. Subroutine COUNTC 14-8
14-8. Sample Terminal Dialog for Subroutine COUNTC 14-9
14-9. Programs MS1 and MS2 14-10
14-10. Program MS4 14-11
14-11. Program SCLCALL 14-12
14-12. Sample Terminal Dialog for Program SCLCALL............... 14-14
E-1. Example of EXAFORT Source Listing........................... E-6

Tables

5-1. Resulting Type for X1 ** X2 5-9
5-2. Resulting Type for X1 + X2, X1 - X2, X1 * X2, or X1 / X2..... 5-10
7-1. FORTRAN Defaults for File Attributes......................... 7-4
7-2. Printer Control Characters 7-10
7-3. Repeatable Edit Descriptors 7-26
7-4. Nonrepeatable Edit Descriptors . 7-27
9-1. Mathematical Intrisic Functions......................... 9-6
10-1. FORTRAN Utility Subprograms.................... 10-38
12-1. VECTOR_FORTRAN Command Parameters............. 12-3
B-1. Related Manuals B-2
C-1. ASCII Character Set and Collating Sequence................. C-2

C-2. OSV$ASCII6_FOLDED Collating Sequence C-6
C-3. OSV$ASCII6_STRICT Collating Sequence C-8
C-4. OSV$COBOL6_FOLDED Collating Sequence C-10
C-5. OSV$COBOL6_STRICT Collating Sequence C-12
C-6. OSV$DISPLAY63_FOLDED Collating Sequence C-14
C-7. OSV$DISPLAY63_STRICT Collating Sequence C-16
C-8. OSV$DISPLAY64_FOLDED Collating Sequence C-18
C-9. OSV$DISPLAY64_STRICT Collating Sequence C-20
C-10. OSV$EBCDIC Collating Sequence C-22
C-11. OSV$EBCDIC6_FOLDED Collating Sequence C-29
C-12. OSV$EBCDIC6_STRICT Collating Sequence C-31
H-1. Edit Descriptors H-10

About This Manual

This manual describes the CONTROL DATA® FORTRAN Version 2 language. FORTRAN Version 2 complies with the American National Standards Institute (ANSI) FORTRAN language described in document X3.9-1978 and known as FORTRAN 77. FORTRAN Version 2 also complies with the International Standards Organization (ISO) standard 1539-1980. The FORTRAN Version 2 compiler is available under the CDC® Network Operating System/Virtual Environment (NOS/VE) operating system.

This manual is intended to be used as a reference. It is assumed that the reader has programmed using the FORTRAN language before.

Audience

To understand this manual, you should be familiar with an existing FORTRAN language. In addition, you should know how to create and run jobs under the NOS/VE operating system. The introductory concepts of NOS/VE are described in the Introduction to NOS/VE manual. A more complete discussion is provided in the NOS/VE System Usage manual.

Organization

The FORTRAN Version 2 manual set consists of the following manuals:

FORTRAN for NOS/VE Tutorial (publication number 60485912)

This manual is intended for the programmer who has no previous FORTRAN experience. It presents a tutorial introduction to the FORTRAN language, beginning with the basic elements of the language and proceeding through more complex features. This manual describes FORTRAN Version 1, although most material applies to FORTRAN Version 2.

FORTRAN for NOS/VE Topics for FORTRAN Programmers (publication number 60485916)

This manual is intended for experienced FORTRAN programmers who are new to NOS/VE. It presents introductory topics intended to help FORTRAN programmers use the NOS/VE operating system and NOS/VE FORTRAN effectively. Topics covered include the System Command Language (SCL), debugging, input/output, optimization, virtual memory, and object libraries. This manual describes FORTRAN Version 1, although most material applies to FORTRAN Version 2.

Online FORTRAN Version 2 for NOS/VE Quick Reference (online manual name VFORTRAN)

This manual provides a quick reference for the FORTRAN Version 2 commands, statements, functions, and subprograms. Parameter descriptions and examples are included. This manual also describes the FORTRAN Version 2 compilation diagnostics. To read this manual, enter

```
/help manual=vfortran
```

This manual is only available in online form.

FORTRAN Version 2 for NOS/VE Language Definition (publication number 60487113)

This manual provides detailed descriptions, definitions, and examples of all the statements and features of the NOS/VE FORTRAN Version 2 language.

FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces (publication number 60485917)

This manual provides detailed descriptions of the keyed-file and Sort/Merge interfaces to FORTRAN. The keyed-file interface allows you to perform input/output operations on keyed files. The sort/merge interface allows you to arrange records in a sequence you specify.

FORTRAN for NOS/VE LIB99 (publication number 60485917)

This manual provides detailed descriptions of the LIB99 library of subroutines and functions. The subroutines and functions perform basic vector arithmetic and matrix algebra, compute Fast Fourier Transforms and eigenvalues and eigenvectors, and solve linear system of equations among other things.

Conventions

Certain notations are used throughout this manual. The notations are:

UPPERCASE	In language syntax, uppercase letters indicate a statement keyword or character defined by FORTRAN for specific purposes. You must use these words exactly as shown. Although lowercase letters are interpreted the same as uppercase characters when used in FORTRAN keywords and symbols, uppercase is used in this manual for consistency. In occasional examples, keywords and symbols are shown in lowercase for illustrative purposes.
lowercase	In language syntax, lowercase letters indicate a name, number, symbol, or entity you choose.
Boldface	In language syntax, boldface type indicates a required keyword, parameter, or symbol.
Italics	In language syntax, italic type indicates optional keywords, parameters, and symbols.
...	In language syntax, a horizontal ellipsis indicates that the preceding optional item can be repeated as necessary.
⋮	In program examples, a vertical ellipsis indicates that other FORTRAN statements or parts of the program have not been shown because they are not relevant to the example.
Δ	Space character. This symbol is used in input/output examples wherever there might otherwise be doubt as to how many spaces are intended.
|	In examples of formatted input and output, vertical bars denote the input or output fields. When used to enclose a numeric quantity, vertical bars indicate the magnitude (absolute value) of the quantity.
Vertical bar	A vertical bar in the margin indicates a technical change from the previous revision of the manual.
Numbers	All numbers are decimal unless otherwise indicated. Other number systems are indicated by a notation after the number; for example, 177 octal or FA34 hexadecimal.

Control Data Extension

Control Data Extensions

Major descriptions of Control Data extensions to standard ANSI FORTRAN are delimited by bars labeled Control Data Extension. Other descriptions of Control Data extensions are shaded. Appendix G lists all Control Data extensions.

End of Control Data Extension

Submitting Comments

There is a comment sheet at the back of this manual. You can use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report errors. Mail your comments to:

Control Data Corporation
Technical Publications
P. O. Box 3492
Sunnyvale, California 94088-3492

Please indicate whether you would like a response.

If you have access to SOLVER, the Control Data online facility for reporting problems, you can use it to submit comments about this manual. When entering your comments, use FV8 as the product identifier. Include the name and publication number of the manual.

If you have questions about the packaging and/or distribution of a printed manual, write to:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

or call (612) 292-2101. If you are a Control Data employee, call (612) 292-2100 or CONTROLNET® 243-2100.

CYBER Software Support Hotline

Control Data's CYBER Software Support maintains a hotline to assist you if you have trouble using our products. If you need help not provided in the documentation, or find the product does not perform as described, call us at one of the following numbers. A support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

Introduction to NOS/VE FORTRAN 1

Standard FORTRAN Capabilities 1-1
Control Data Extensions 1-2
Input and Output of the FORTRAN Compiler 1-4
The NOS/VE Environment 1-6

Introduction to NOS/VE FORTRAN 1

NOS/VE FORTRAN provides the features and capabilities set forth in the ANSI and ISO FORTRAN Standard as well as several unique Control Data features.

Standard FORTRAN Capabilities

NOS/VE FORTRAN Version 2 offers the full complement of standard FORTRAN capabilities. These capabilities include integer, single precision, double precision, and complex arithmetic; block control structures; character string processing; and a wide range of input/output capabilities.

NOS/VE FORTRAN Version 2 (hereafter referred to simply as FORTRAN) can use the vectorization capabilities of the CYBER 990 or 995 class mainframes to improve the execution time of your program. Your program can also be compiled and executed on other machines. (See the TARGET_MAINFRAME parameter on the VECTOR_FORTRAN command.)

FORTRAN programs that strictly comply with the ANSI standard can be compiled using the FORTRAN compiler and executed under NOS/VE with no changes, regardless of the computer system for which the program was originally written.

Control Data Extensions

In addition to the standard features, NOS/VE FORTRAN provides unique features which greatly enhance the power of the FORTRAN language. Most of these features consist of subprogram calls that allow you to take advantage of other Control Data products.

Although a program that uses these features cannot be transported directly to another computer system, these extensions allow greater flexibility and choice of options when you write FORTRAN programs. To migrate programs from one system to another, a compiler option is available that detects most non-ANSI usages within a program.

The major extensions include:

Symbolic name length

Allows symbolic names to be from 1 to 31 characters and contain underscores and dollar signs.

Variable-length integer, real, logical, and complex data

Allows 2-, 4-, or 8-byte integer values; 8- or 16-byte real values; 1-, 2-, 4-, or 8-byte logical values; or 16-byte complex values. These are intended to provide compatibility with other versions of FORTRAN. The type statements used to declare these values are described in chapter 4, Specification Statements.

Byte data type

Allows signed 1-byte integer values. The byte data type is intended to provide compatibility with other versions of FORTRAN. The BYTE type statement is described in chapter 4, Specification Statements.

Boolean data type

Allows you to manipulate octal, hexadecimal, and boolean string data items. Boolean constants and variables are described in chapters 2 and 4; boolean expressions are described in chapter 5, Expressions and Assignment Statements.

Array sections

Allows you to access and modify groups of elements in an array by referencing a section of an array. You can use array section references in assignment, WHERE, and input/output statements. Array sections are described in chapter 3, Arrays.

Namelist input/output

Allows you to perform formatted input/output without specifying a format or an input/output list. You specify a group name, and all items in the group are read or written according to a compiler-defined format. Namelist input/output is described in chapter 7, Input/Output.

Segment access files

Allows you to reference files mapped to a named common block in the same manner as normal variables in a named common block. Segment access files are described in chapter 7, Input/Output, and chapter 11, C$ Directives.

Mass storage input/output

Allows you to create and access random files. Records in random files are accessed directly by record key. This provides a quicker method of access than conventional sequential input/output. Mass storage input/output is described in chapter 7, Input/Output.

Array-processing intrinsics

Allows you to perform various operations on arrays, such as determining the upper or lower bound, largest or smallest element, dot product, number of dimensions, or the size of the array. The array-processing intrinsic functions are described in chapter 9, Intrinsic Functions.

C$ directives

Allow you to control various aspects of compilation, such as whether or not certain source lines are to be compiled or ignored by the compiler or whether or not certain source lines are not eligible for vectorization. C$ directives are described in chapter 11, C$ Directives.

Descriptions of Control Data extensions are shaded or enclosed in bars labeled Control Data Extension. Appendix G, Control Data Extensions, lists all Control Data extensions to ANSI FORTRAN.

Input and Output of the FORTRAN Compiler

The FORTRAN compiler reads a file containing a FORTRAN source program, translates the program into an object program consisting of machine instructions, and (optionally) writes the object program to a file. The object program can then be loaded into memory and executed by NOS/VE commands.

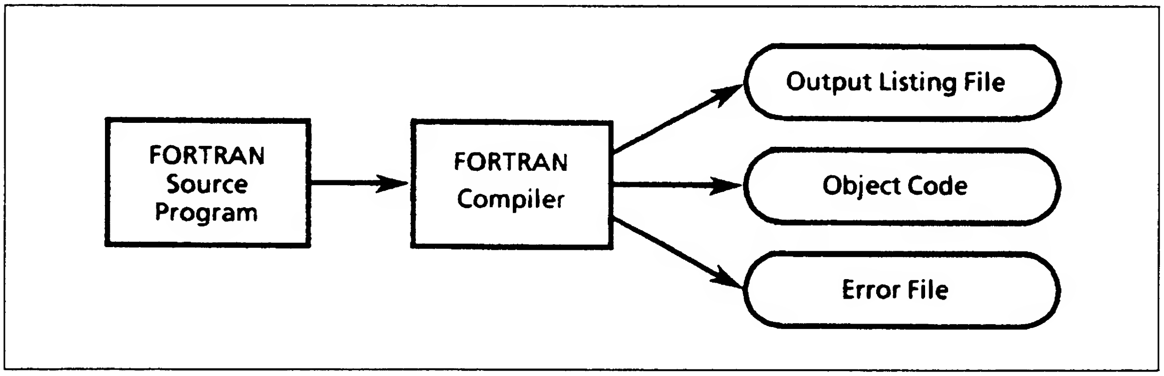


Figure 1-1. The FORTRAN Compiler

A FORTRAN source program consists of text lines formatted according to the rules of FORTRAN syntax. If the compiler detects a syntax error in the source program, it issues a descriptive message describing the nature of the error. The compiler detects errors at different levels of severity. If the errors are severe enough, the resulting object program cannot be executed; you must correct the errors and recompile.

Generally, the compiler diagnostic messages provide enough information to enable you to easily determine the cause of the errors. For more information on a compiler diagnostic message, see the VFORTRAN online manual. Use the error number as an index topic. For example, to read about compiler diagnostic FV2010, enter:

```
/help manual=vfortran subject='2010'
```

The FORTRAN Version 2 compiler produces object code at two levels of optimization: low and high. The lower level results in faster compilation, but produces an object program that executes slower. At the higher level of optimization, the compiler manipulates the generated object code to produce an object program that executes much faster. The compiler also produces object code modified for debugging at the low level of optimization.

The level of optimization is selected by a parameter on the compilation command. You can generate object code at any level. You can also specify a compiler option to take advantage of the CYBER 990 or 995 class mainframes for vectorization. An optional implicit vectorization report provides messages indicating the level of vectorization accomplished on your program.

In addition to the object program, the FORTRAN compiler produces two other output files: an output listing file and an error listing file. These files are optional and are selected by parameters on the compilation command. The error listing file contains error messages that were issued during compilation. The output listing file contains a complete listing of the source program and, optionally, an object listing and a reference map. The reference map provides detailed information about symbolic names and other items used in the FORTRAN program and is a useful debugging tool.

The FORTRAN compiler provides a number of other options in addition to those described above. The available compiler options, the formats of the input and output files, and the commands for compiling and executing a FORTRAN program are described in chapter 12. Vectorization concepts and the vectorization report are described in chapter 13.

The NOS/VE Environment

The NOS/VE operating system provides several software facilities that can make creation and maintenance of FORTRAN programs easier and more efficient. The following facilities can be used outside your program:

Source Code Utility (SCU)

Creates, updates, and manipulates libraries of source programs. SCU is described in the NOS/VE Source Code Management manual.

Object Code Management Utilities

Creates and maintains libraries of compiled object programs (called object libraries). Object libraries allow programs to share compiled subprograms. The MEASURE_PROGRAM_EXECUTION utility can be used to measure and analyze program performance. The AFTERBURN_OBJECT_TEXT command can be used to expand FORTRAN routines inline. These utilities are described in the NOS/VE Object Code Management manual.

File Migration Aid (FMA)

Reads, writes, and edits ANSI standard FORTRAN files on the NOS or NOS/BE side of a dual-state system from your FORTRAN program running on NOS/VE. FMA is described in the Migration from NOS to NOS/VE manual.

File Management Utility (FMU)

Allows you to convert data records from one format to another. FMU can convert data files from other systems for use on NOS/VE. FMU is described in the NOS/VE Advanced File Management manual.

Debug Utility

Debugs a program during its execution. You can stop the program at selected points or on the occurrence of an error and request formatted displays of variables and arrays. The Debug utility is described in the Debug for NOS/VE manual. A brief introduction is presented in appendix E.

Programming Environment (PE)

Allows you to create, debug and run FORTRAN programs in an integrated environment using a screen interface. You can access the editor, the Debug utility, and the online manuals. The Programming Environment is described in the online ENVIRONMENT manual. A brief introduction is presented in appendix F.

Professional Programming Environment (PPE)

Coordinates complex programming projects using an integrated screen environment. You can access the editor, the Debug utility, and the online manuals. PPE coordinates multiple levels of code development using SCU and object libraries. The Professional Programming Environment is described in the Professional Programming Environment manual. A brief introduction is presented in appendix F.

The following facilities can be used from within your FORTRAN program.

System Command Language Calls

These calls provide a method of communicating with the operating system using the System Command Language (SCL). The parameter interface calls can reference the parameters on the command that began execution of the FORTRAN program. The variable interface calls can retrieve or alter the values of existing SCL variables, as well as define new SCL variables. The SCLCMD call can execute any NOS/VE command from within your program. These calls are described in chapter 10, NOS/VE and Utility Subprograms.

General Utility Calls

These calls enable you to perform a variety of tasks, such as generating program dumps, generating random number sequences, and obtaining time and date information from the system. These calls are described in chapter 10, NOS/VE and Utility Subprograms.

Sort/Merge Calls

These calls enable you to sort the records of one or more files into a specific order and to merge the sorted records of two or more files into a single file. These calls are described in the FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces manual.

Keyed-File Calls

These calls enable you to use the keyed-file organizations (indexed sequential and direct access) in your FORTRAN program. These calls are described in the FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces manual.

IM/DM

You can place IM/DM commands in your FORTRAN program. If you do, the program must be processed by the IM/DM preprocessor before compilation occurs. The DM Concepts and Facilities manual describes the commands used to manipulate an IM/DM database and the utilities available to application programmers.

Language Elements 2

The FORTRAN Character Set 2-2

FORTRAN Statements 2-4
  Comments 2-4
  Initial Lines 2-5
  Continuation Lines 2-5
  Statement Labels 2-5
  Compiler Directive Lines 2-5
  Positions 73 and Beyond 2-6

Statement Order 2-7

Symbolic Names 2-9

Constants 2-10
  Integer Constants 2-11
  Real Constants 2-12
    Range of Real Constants 2-13
  Character Constants 2-14
  Boolean Constants 2-15
    Hollerith Constants 2-15
    Octal Constants 2-16
    Hexadecimal Constants 2-17
    Extended Hollerith Constants 2-18
  Complex Constants 2-20
  Double Precision Constants 2-21
  Logical Constants 2-22

Variables 2-23
  How to Declare the Type of Variables 2-23
  Lengths of Variables 2-23

Language Elements 2

FORTRAN statements are composed of elements that are combined according to the rules of FORTRAN syntax. These elements include constants, variables, substrings, and operators. These elements, and other FORTRAN constructs, are named using the FORTRAN character set.

Constants, variables, and substrings are all scalar types. A scalar is a type that represents a single value; an array is a group of scalars. Arrays are described in chapter 3.

The FORTRAN Character Set

FORTRAN statements are written using the FORTRAN character set shown in the following table:

Type	Characters	Meaning
Alphabetic	A through Z	(Lowercase letters are equivalent to uppercase letters when used in symbolic names and FORTRAN reserved words)
Numeric	0 through 9	
Special Characters	=	equals
	+	plus
	-	hyphen
	*	asterisk
	/	slant
	(	opening parenthesis
	)	closing parenthesis
	,	comma
	.	period
	'	apostrophe
	:	colon
	"	quotation marks
	!	exclamation point
	_	underscore
	$	dollar sign[1]
		space

The ASCII representations of the characters are shown in appendix C.

Lowercase letters are equivalent to uppercase letters when used in symbolic names and FORTRAN keywords. Symbolic names are described later in this chapter. The following FORTRAN statements are equivalent:

```
READ (UNIT=1,FMT=99) AVAL, Z

read (unit=1,fmt=99) aval, z
```

1. NOS/VE often uses $ as the fourth character in its predefined names. To keep from matching a system-reserved name, avoid using $ as the fourth character in the names you define.

However, in character strings, boolean string constants, and extended Hollerith constants, uppercase and lowercase characters are treated as distinct values. The following character constants are not equivalent:

```
'ABCDE'

'abcde'
```

NOTE

For consistency and readability, the FORTRAN syntax descriptions in this manual use uppercase letters to indicate keywords and lowercase letters to indicate user-supplied values. However, in all FORTRAN statements, lowercase letters are valid and are treated the same as uppercase letters.

ASCII characters that are not included in the FORTRAN character set can be used in:

Character strings

Boolean strings

Extended Hollerith constants

Apostrophe, quote, and H edit descriptors

Comment lines

Inline comments

FORTRAN Statements

A FORTRAN statement is written on one or more lines. The first (and possibly only) line of each statement is an initial line. Each additional line is a continuation line. Each statement has one initial line and can have 0 through 19 continuation lines.

Lines can also be comment or compiler directive lines.

A line of a FORTRAN statement consists of characters in positions 1 through 80. However, only positions 1 through 72 are scanned by the compiler. You can use positions 73 and beyond for an identification field. The following example shows a simple FORTRAN program:

```
      PROGRAM PASCAL
C
C  THIS PROGRAM PRODUCES A PASCAL TRIANGLE
C
      INTEGER LAST_ROW(15)
      LAST_ROW = 1
      PRINT '(17H1 PASCAL TRIANGLE, //1X,I5,/1X,2I5)',
     +LAST_ROW(15), LAST_ROW(14), LAST_ROW(15)
      DO 50 J = 14,2,-1
         DO 40 K = J,14
            LAST_ROW(K) = LAST_ROW(K) + LAST_ROW(K+1)
 40      CONTINUE
         PRINT '(1X,15I5)', (LAST_ROW(M), M = J-1,15)
 50   CONTINUE
      END
```

Figure 2-1. FORTRAN Program Example

Comments

Comments provide a method of placing program documentation in the source program. Comments can appear as inline comments or as entire comment lines.

An inline comment is written on the same line as a FORTRAN statement. An exclamation point (!) terminates the FORTRAN line and marks the beginning of an inline comment. However, an exclamation point (!) appearing in column 6 marks a continuation line and an exclamation point (!) appearing in column 1 marks an entire comment line.

The appearance of an inline comment does not affect the preceding FORTRAN statement.

A comment line is indicated by a C, an asterisk (*), or an exclamation point (!), in position 1. Comment lines do not affect the program and can be placed anywhere within the program. Comment lines can appear between an initial line and a continuation line, or between two continuation lines. A comment line following an END statement is treated as the first line of the next program unit (program units are described in chapter 8).

Any line with spaces in positions 1 through 72 is also a comment line.

Additional characters that are not in the FORTRAN character set can be included in comment lines. Comment lines can include any printable graphic character listed in appendix C for the character set being used.

Initial Lines

Each statement begins with an initial line. The statement characters are written in positions 7 through 72 of the initial line. You can use spaces to improve readability. The initial line of a statement can contain a statement label in positions 1 through 5. Position 6 must be a space or a zero.

Continuation Lines

If a statement is longer than 66 characters (positions 7 through 72 of the initial line), it can be continued on as many as 19 continuation lines. A character other than a space or zero in position 6 indicates a continuation line. Positions 1 through 5 must contain spaces.

The length of a statement cannot exceed 1320 characters (one initial line and 19 continuation lines, at 66 characters per line).

Statement Labels

A statement label must be a 1- through 5-digit integer. The integer must be positive and not equal to zero. The label is written in positions 1 through 5 of the initial line of a statement. A statement label uniquely identifies a statement so that it can be referenced by other statements. Any statement can be labeled, but only FORMAT and executable statement labels can be referenced by other statements. Statements that will not be referenced do not need labels. Spaces and leading zeros are not significant.

Labels need not occur in numerical order, but a given label must not be declared more than once in the same program unit. A label is known only in the program unit containing it and cannot be referenced from a different program unit.

Control Data Extension

Compiler Directive Lines

A compiler directive is a special form of comment line that affects compiler behavior.

The characters C and $ in positions 1 and 2 indicate a compiler directive line. A compiler directive must appear on a single line and any compiler directive terminates statement continuation.

Compiler directives are effective unless the COMPILATION_DIRECTIVES parameter of the VECTOR_FORTRAN command specifies OFF, in which case, compiler directives are interpreted as comment lines.

Each directive, including keyword and parameters, is written in positions 7 through 72. Compiler directives are described in chapter 11.

End of Control Data Extension

Positions 73 and Beyond

Positions 73 and beyond can be used for identification information. Characters in the identification field are ignored by the compiler but are copied to the source program listing.

Statement Order

The order of various statements within the program unit is shown in the following diagram.

Within each group, you can order statements as necessary, but you must order the groups as shown. Statements that can appear anywhere within more than one group are shown on the right in boxes that extend vertically across more than one group.

A PROGRAM statement can appear only as the first statement in a main program. The first statement of a subroutine, function, or block data subprogram is respectively, a SUBROUTINE statement, FUNCTION statement, or BLOCK DATA statement.

Comments can appear anywhere within a program unit. Note that a comment following the END statement is considered part of the next program unit. FORMAT statements can also appear anywhere in a program unit. ENTRY statements can appear anywhere in a program unit, subject to three restrictions; an ENTRY statement cannot appear within:

- the range of a DO loop (between the DO statement and the final statement of the DO loop)
- a block IF structure (between the IF statement and the ENDIF statement)
- a block WHERE structure (between the WHERE statement and the ENDWHERE statement)

The ENTRY statement cannot be used in the main program unit.

Specification statements in general precede the executable statements in a program unit. The nonexecutable specification statements describe characteristics of quantities known in the program unit; executable statements describe the actions to be taken. All specification statements must precede all DATA statements, NAMELIST statements, statement function definitions, and executable statements. Within the specification statements, all IMPLICIT statements must precede all other specification statements except PARAMETER statements. PARAMETER statements can appear anywhere among the specification statements, but each PARAMETER statement must precede any references to symbolic constants defined by that PARAMETER statement.

All statement function definitions must precede all executable statements in a program unit. Statement function definitions must not appear in block data subprograms. DATA statements can appear anywhere among statement function definitions and executable statements.

NAMELIST statements can appear anywhere among statement function definitions and executable statements. Each NAMELIST statement defining a namelist group must appear before the first reference to that namelist group. Also, NAMELIST statements must not appear in block data subprograms.

Executable statements must follow all specification statements and any statement function definitions. Executable statements such as assignment, flow control, or I/O statements, can appear in whatever order required in the program unit. Executable statements cannot appear in block data subprograms.

The END statement must be the last statement of each program unit.

<table>
<tr><th colspan="6">Statement</th></tr>
<tr><td colspan="5">PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA</td><td rowspan="7">Comments
and
compiler
directives</td></tr>
<tr><td colspan="2">IMPLICIT</td><td rowspan="3">PARAMETER
(must
precede
first
reference)</td><td rowspan="6">FORMAT[1]</td><td rowspan="6">ENTRY[2]
(except
within
range
of block
IF or DO
loop)</td></tr>
<tr><td colspan="2">BOOLEAN
BYTE
CHARACTER
COMPLEX
DOUBLE PRECISION
INTEGER
LOGICAL
REAL
} Type specification statements</td></tr>
<tr><td colspan="2">COMMON
DIMENSION
ENDINTERFACE
EQUIVALENCE
EXTERNAL
INTERFACE
INTRINSIC
SAVE
} Specification statements</td></tr>
<tr><td>Statement function definition[1]</td><td rowspan="3">DATA</td><td rowspan="3">NAMELIST[1]
(must
precede
first
reference)</td></tr>
<tr><td>ALLOCATE
ASSIGN
Assignment
CALL
CONTINUE
DEALLOCATE
DO
ELSE
ELSEIF
ELSEWHERE
ENDIF
ENDWHERE
GOTO
IF
PAUSE
RETURN
STOP
WHERE
} Executable[1] statements</td></tr>
<tr><td>BACKSPACE
BUFFER IN
BUFFER OUT
CLOSE
DECODE
ENCODE
ENDFILE
INQUIRE
OPEN
PRINT
PUNCH
READ
REWIND
WRITE
} Executable[1] I/O statements</td></tr>
<tr><td colspan="6">END</td></tr>
<tr><td colspan="6">1. Cannot be used in a BLOCK DATA subprogram.
2. Cannot be used in a main program or BLOCK DATA subprogram.</td></tr>
</table>

Symbolic Names

A symbolic name consists of 1 through 31 letters, digits, dollar signs, or underscores, beginning with a letter (ANSI only allows 6 letters and digits). Lowercase letters are equivalent to uppercase letters. Spaces in a symbolic name are ignored. NOS/VE often uses $ as the fourth character in its predefined names. To keep from matching a system-reserved name, avoid using $ in the fourth position of names you define.

Symbolic names are used for the following:

Symbolic constant name
Variable name
Array name
Common block name
DO-variable name in an implied DO list in a DATA statement
Namelist group name
Main program name
Subroutine name
Statement function name
External function name
Intrinsic function name
Block data subprogram name
Dummy subprogram name

Throughout this manual, symbolic names are referred to as names.

You can use names that are FORTRAN keywords as symbolic names without conflict. For example:

```
PROGRAM TEST
PRINT = 1.0
PRINT*, PRINT
      ⋮
```

The name PRINT is legally used as both a variable name and a FORTRAN keyword.

However, certain naming conflicts are illegal and are diagnosed. For example, the following sequence illegally uses the name ALPHA as a program unit name and a variable name:

```
PROGRAM ALPHA
ALPHA = 1.0
      ⋮
```

The following example illegally uses the name STING_RAY as both an array name and a subroutine name:

```
PROGRAM X
DIMENSION STING_RAY(3)
      ⋮
CALL STING_RAY
      ⋮
```

In general, you should avoid naming conflicts by assigning unique names to all program entities.

Constants

A constant is a scalar quantity that remains fixed throughout program execution. The types of constants are

Integer
Real
Double precision
Complex
Boolean
Logical
Character

You can reference a constant by its actual value or, with the exception of extended Hollerith constants, by a symbolic name associated with the constant. You can use the PARAMETER statement (described in chapter 4) to assign a symbolic name to a constant. Integer, byte, real, double precision, complex, and boolean constants are considered arithmetic constants.

Spaces in a constant, except a character, boolean string, or extended Hollerith constant, have no effect on the value of the constant.

Integer Constants

An integer constant is a string of 1 through 19 decimal digits with no decimal point. An integer constant has the form:

± **d**...*d*

d

Decimal digit

An integer can be positive, negative, or zero. If the integer is positive, the plus sign can be omitted; if it is negative, the minus sign must be present. An integer constant must not contain a comma.

Integer constants are usually 8 bytes long; however, symbolic constants of type integer can be declared with a length of 2, 4, or 8 bytes. You can also declare a symbolic constant of type byte, which is a 1-byte integer. See the PARAMETER statement in chapter 4, Specification Statements, for more information on symbolic constants.

Examples of valid integer constants:

```
237
-74
+136772
-0024
```

Examples of invalid integer constants:

`46.`	Decimal point not allowed
`23A`	Letter not allowed
`7,200`	Comma not allowed

NOTE

Throughout this manual, whenever an integer constant, variable, or expression is allowed, it can be of any length or be of data type byte, unless otherwise noted.

Real Constants

A real constant is a string of decimal digits with a decimal point or an exponent or both. A real constant has one of the forms:

± **coeff**

± **coeff E** ± **exp**

± **n E** ± **exp**

coeff

Coefficient in one of the forms:

n.
n.n
.n

where

n

Unsigned integer constant

exp

Unsigned integer exponent (base 10). This is the power of ten by which the number is multiplied.

If the exponent E (or e) is present, the integer constant following the letter E must also be present. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

Real constants are usually 8 bytes long; however symbolic constants of type real can be declared with a length of 8 or 16 bytes.

Sixteen-byte real constants are treated as double precision values.

See the PARAMETER statement in chapter 4, Specification Statements, for more information on symbolic constants.

Examples of valid real constants:

7.5	Value is 7.5.
-3.22	Value is -3.22
+4000.	Value is 4000.
.5	Value is .5.
42E1	Value is 42. x 10**1 = 420.
.000285E+5	Value is .000285 x 10**5 = 28.5
6.205E6	Value is 6.205 x 10**6 = 6,205,000.
14.D-5	Value is 14. x 10**(-5) = .00014
700.e-2	Value is 700. x 10**(-2) = 7. (the symbol e is equivalent to E)

Examples of invalid real constants:

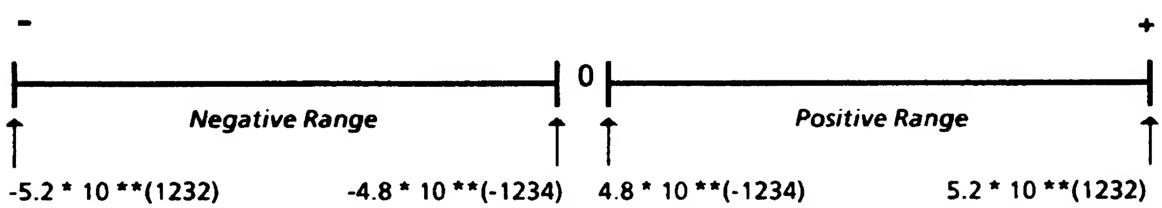
33,500.	Comma not allowed
2.5A	Letter not allowed
7.2E-3.4	Exponent is not an integer

Range of Real Constants

The range of real constants of either length is as follows:

```
-5.2 * 10**(1232) through -4.8 * 10(-1234)
0.
4.8 * 10**(-1234) through 5.2 * 10**(1232)
```

These ranges are illustrated on the following number line:



Note that the above number line is not to scale but it indicates that real numbers approach 0 from the negative boundary -4.8 * 10**(-1234) and from the positive boundary 4.8 * 10**(-1234).

A number other than 0 between -4.8 * 10**(-1234) and 4.8 * 10**(-1234) generates an exponent underflow error.

A number less than -5.2 * 10**(1232) or greater than 5.2 * 10**(1232) generates an exponent overflow error.

Numbers beyond -5.2 * 10**(1232) and 5.2 * 10**(1232) are defined as negative and positive infinity, respectively.

Character Constants

A character constant is a string of printable characters enclosed by apostrophes. A character constant has the form:

'**s**...*s*'

s

Any printable ASCII character

Apostrophes(') are used to enclose the character string. Within the character string, an apostrophe is represented by two consecutive apostrophes(''). The length is the number of characters in the string; however, the two consecutive apostrophes count as 1 character in the length of the string. Spaces are significant in a character constant.

The minimum number of characters in a character constant is 1; the maximum number of characters is (2**16)-1 or 65,535. The string can contain any printable graphic character in the ASCII set. An uppercase character is not equivalent to its lowercase counterpart.

Character positions in a character constant are numbered consecutively from the left as 1, 2, 3, and so forth, up to the length of the constant. The length of the character constant is significant in all operations in which the constant is used. The length must be greater than zero.

Examples of valid character constants:

`'ABC'`	Length 3
`'123'`	Length 3
`'Year''s'`	Represents the string Year's and has length 6

Examples of invalid character constants:

`'ABC`	Terminating apostrophe is missing.
`"ABC"`	Double quotes are used.
`'Year's'`	Apostrophes within string must be doubled.

Control Data Extension

Boolean Constants

A boolean constant is a Hollerith constant, an octal constant, a hexadecimal constant, or an extended Hollerith constant. (An extended Hollerith constant is used when an actual argument is more than 8 characters.) A boolean constant is represented in one computer word (8 bytes).

Hollerith Constants

A Hollerith constant has one of the following forms:

nHs

L"s"

R"s"

"s"

n

An unsigned nonzero integer constant in the range 1 through 8

s

A string of 1 through 8 characters

A Hollerith constant can usually contain no more than 8 characters. Extra characters are truncated on the right. However, an extended Hollerith constant (a Hollerith constant that is used as an actual argument) or a Hollerith constant in a format specifications can exceed 8 characters.

The meaning of each form is as follows:

nHs and "s" forms

Indicate that characters are left-justified with blank fill. Blank fill means that any unassigned character positions in the computer word are set to space (ASCII code 20 hexadecimal). The value **n** in **nHs** specifies the number of characters in the string **s**.

Example:

`2HAB`	Value is 414220...20 hexadecimal.

L"s" form

Indicates left-justified with binary zero fill. Binary zero fill means any unassigned character positions are set to binary zero (ASCII code 00 hexadecimal).

Example:

`L"AB"`	Value is 414200...00 hexadecimal

Control Data Extension *(Continued)*

R"s" form

Indicates right-justified with binary zero fill. Binary zero fill means any unassigned character positions are set to binary zero (ASCII code 00 hexadecimal). Example:

`R"AB"`	Value is 00...004142 hexadecimal

"s" form

is equivalent to the nHs form except that the characters need not be counted.

In the **L"s"**, **R"s"**, and **"s"** forms, a quote within the string is represented by two consecutive quote characters (""); the consecutive quotes count as 1 character. In all forms, spaces are significant and any printable graphic character listed in appendix C for the character set in effect can be used.

Examples:

`"AB"`	Value is 414220...20 hexadecimal.
`"C""D"`	Value is 43234420...20 hexadecimal.

Octal Constants

An octal constant has the form:

O"o"

o

A string of 1 through 22 octal digits.

An octal digit is one of the digits: 0, 1, 2, 3, 4, 5, 6, or 7. The string of octal digits is interpreted as an octal number. As many as 22 octal digits can be represented in a computer word. If all 22 digits are used, the leftmost digit must be 0 or 1. The octal number is right-justified with binary zero fill. Binary zero fill means any unassigned character positions are set to binary zero (ASCII code 00 hexadecimal).

Example:

`O"77"`	Value is 00...003F hexadecimal.

Control Data Extension *(Continued)*

Hexadecimal Constants

A hexadecimal constant has the form:

Z"z"

z

A string of 1 through 16 hexadecimal digits.

A hexadecimal digit is one of the characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, or F. Lowercase letters are allowed. The string of hexadecimal digits is interpreted as a base 16 number. As many as 16 hexadecimal digits can be represented in a computer word. The hexadecimal number is right-justified with binary zero fill. Binary zero fill means any unassigned character positions are set to binary zero (ASCII code 20 hexadecimal).

Examples:

`Z"1A"`	Value is 00...001A hexadecimal.
`Z"ace"`	Value is 00...ACE hexadecimal.

Control Data Extension *(Continued)*

Extended Hollerith Constants

Extended Hollerith constants are used only as actual arguments to external procedures. An extended Hollerith constant has one of the following forms:

nHs

L"s"

R"s"

"s"

n

An unsigned nonzero integer constant greater than eight that specifies the number of characters in string **s**.

s

A string of **n** characters for the **nHs** form, or a string of greater than 8 characters for any of the other forms.

An extended Hollerith constant is stored in two or more consecutive computer words. The length in words of an extended Hollerith constant is given by the expression

INT((N + 8 - 1)/8)

where N is the number of characters in the constant. (INT is an intrinsic function that truncates the fractional part of the result of the division.)

An extended Hollerith constant is stored beginning in the leftmost character position of the first word. If there are any unassigned character positions in the last word occupied by the constant, those positions are filled with either spaces or zeros depending on the form of the constant. The characters in the last word occupied by the constant are either left-justified or right-justified within the word depending on the form of the constant.

In an extended Hollerith constant, spaces are significant and any printable graphic character not in the FORTRAN character set can be used.

For the **nHs** and **"s"**forms, characters in the last word occupied by the constant are left-justified with blank fill. Blank fill means that any unassigned character positions in the last word occupied by the constant are set to space characters (ASCII code 20 hexadecimal).

Example:

`10HABCDEFGHIJ` Value is 4142434445464748 494A20...0020 hexadecimal

Control Data Extension ***(Continued)***

For the L"s", R"s", and "s"forms, a quote (") character within the string is represented by two consecutive quote (") characters; the consecutive quotes count as 1 character.

For the L"s" form, characters in the last word occupied by the constant are left-justified with binary zero fill. Binary zero fill means that any unassigned character positions in the last word occupied by the constant are set to binary zero (ASCII code 00 hexadecimal).

Example:

`L"ABCDEFGHIJ"`	Value is 4142434445464748 494A00...00 hexadecimal.

For the R"s" form, characters in the last word occupied by the constant are right-justified with binary zero fill. Binary zero fill means that any unassigned character positions in the last word occupied by the constant are set to binary zero (ASCII code 00 hexadecimal).

Example:

`R"ABCDEFGHIJ"`	Value is 4142434445464748 00...00494A hexadecimal.

The "s" form is equivalent to the nH form except that you need not count characters.

Examples:

`"ABCDEFGHIJ"`	Value is 4142434445464748 494A20...20 hexadecimal.
`"QRSTU""VWXYZ"`	Value is 5152535455235657 58595A20...20 hexadecimal (represents the string QRSTU"VWXYZ).

End of Control Data Extension

Complex Constants

A complex constant is written as a pair of real, integer, or symbolic constants separated by a comma and enclosed in parentheses. A complex constant has the form:

(real, imag)

real

Real, integer, or symbolic constant that represents the real part.

imag

Real, integer, or symbolic constant that represents the imaginary part.

NOTE

The term real as applied to the first component of a complex value should not be confused with the FORTRAN real data type.

The first constant represents the real part of the complex number, and the second constant represents the imaginary part. The parentheses are a required part of the constant. Either constant can be preceded by a plus or minus sign. Complex values are represented internally by two consecutive computer words containing real values.

Type real constants that form the complex constant must be within the valid range for real constants.

Examples of valid complex constants (i = square root of -1):

`(1, 7.54)`	Value is 1.0 + 7.54i
`(-2.1E1, 3.24)`	Value is -21.0 + 3.24i
`(4, 5)`	Value is 4.0 + 5.0i
`(0., -1.)`	Value is 0.0 - 1.0i

Examples of invalid complex constants:

`(12.7D-4 16.1)`	Comma is missing, and double precision is not allowed.
`4.7E+2, 1.942`	Parentheses are missing.

Double Precision Constants

A double precision constant is a string of decimal digits written with an exponent prefixed by the letter D (or d). A double precision constant has one of the forms:

± coeff D ± exp

± n D ± exp

coeff

Coefficient in one of the forms:

n.
n.n
.n

where

n

Unsigned integer constant.

exp

Unsigned integer exponent (base 10).

Double precision values are represented internally by two consecutive words, giving additional precision. The range of a double precision constant is essentially that of a real constant. A double precision constant is accurate to approximately 29 decimal digits.

A plus sign preceding the coefficient is optional for a positive constant; the minus sign is required for a negative constant. The plus sign preceding the exponent can be omitted if the exponent is positive, but the minus sign must be present to denote a negative exponent.

Examples of valid double precision constants:

`5.834D2`	Value is 5.834 x 10**2 = 583.4.
`14.D-5`	Value is 14. x 10**(-5) = .00014.
`9.2d03`	Value is 9.2 x 10**3 = 9200 (the symbol d is equivalent to D).
`3120D4`	Value is 3120. x 10**4 = 31,200,000.

Examples of invalid double precision constants:

`7.2D`	Exponent is missing.
`D5`	Exponent alone is not allowed.
`2,001.3D2`	Comma is not allowed.
`3.14159265`	D and exponent are missing.

For Better Performance

Use double precision constants, variables, arrays, and functions only when necessary because they require more execution time because of the extra precision they support (two words).

Logical Constants

A logical constant is a logical value. A logical constant has one of the following values:

.TRUE. Represents the logical value true.

.FALSE. Represents the logical value false.

The periods are required parts of the constant.

Examples of valid logical constants:

```
.TRUE.   .true.   .True.

.FALSE.  .false.  .False.
```

Examples of invalid logical constants:

`.TRUE` No terminating period.

`.F.` Abbreviation not recognized.

The logical value .true. is represented as a word with a leftmost bit of 1. the logical value .false. is represented as a word with a leftmost bit of 0. The rightmost bits of either word are undefined.

Logical constants are usually 8 bytes long; however, symbolic constants of type logical can be declared with a length of 1, 2, 4, or 8 bytes. The value of a 4-byte logical constant is determined by the sign bit. The value of a 2-byte logical constant is determined by the value of the 2 bytes. Similarly, the value of a 1-byte logical constant is determined by the value of the 1 byte. See the PARAMETER statement for more information on symbolic constants.

Variables

A variable represents a scalar value that can be changed repeatedly during program execution. A variable is identified by a symbolic name. A value must be assigned to a variable before the variable is referenced. The types of variables are:

Integer
Byte
Real
Double precision
Complex
Boolean
Logical
Character

How to Declare the Type of Variables

By default, variables are typed according to the first letter of the variable name. A variable is of type integer if the first letter is I, J, K, L, M, or N. A variable is of type real if the first letter of its name is any other letter. You can change the default type by implicit or explicit typing.

Variables are explicitly or implicitly declared to be of a certain type by appearing in one of the following specification statements:

BOOLEAN
BYTE
CHARACTER
COMPLEX
DOUBLE PRECISION
IMPLICIT
INTEGER
LOGICAL
REAL

These statements are described in chapter 4, Specification Statements.

Lengths of Variables

Integer, logical, and complex variables can be explicitly typed with specific lengths. This ability is provided for compatibility with other versions of FORTRAN. Length specifications are described in chapter 4, Specification Statements.

Arrays 3

Declaring Arrays 3-1

Declaring Allocatable Arrays 3-5
  ALLOCATE Statement 3-5
  DEALLOCATE Statement 3-6

Array Shape and Conformability 3-7

Array Storage 3-8

Array References 3-10
  Complete Arrays 3-10
  Array Elements 3-11
  Array Sections 3-14
  Substrings in Arrays 3-17

Arrays 3

An array is a sequence of elements identified by a single name. You reference each element in the array by the array name and a subscript. The type of an array is determined by the array name in the same manner as the type of a variable is determined by the variable name.

An array name can be typed explicitly with a type statement, implicitly with an IMPLICIT statement, or by default typing. The array name and its dimensions must be declared in a COMMON, DIMENSION, or type statement. These statements are described in chapter 4, Specification Statements.

Declaring Arrays

An array declaration must appear in a COMMON, DIMENSION, or type statement. An array declaration has the form:

array(d, ..., *d*)

array

Array name.

d

Specifies the bounds of an array dimension in one of the forms:

lower : **upper**

lower **:**

where:

lower

Specifies the lower bound of the dimension. The lower bound can be an integer, byte, or boolean expression with a positive, zero, or negative value. If omitted, the lower bound defaults to 1.

upper

Specifies the upper bound of the dimension. The upper bound can be an integer, byte, or boolean expression with a positive, zero, or negative value. For an assumed-size array, the upper bound of the last dimension must be specified as *.

The number of array dimension bounds specified indicates the number of dimensions in the array. Arrays can have from one through seven dimensions.

The upper bound is usually greater than or equal to the lower bound. The size of each dimension is the distance between the lower and upper bound, that is, MAX(*upper-lower+1, 0*) where *upper* is the value of the upper dimension bound and *lower* is the value of the lower dimension bound. If the upper bound is less than the lower bound, the size of that dimension is zero, and the size of the array is zero.

An expression defining a dimension bound can contain symbolic constants defined in previous PARAMETER statements; integer symbolic constants can be of any length. A dimension bound expression in a function or subroutine can contain dummy arguments.

An expression defining a dimension bound must not contain nonintrinsic function, array section, or array element references. Arguments to intrinsic functions can contain other intrinsic function references, integer variables, constants, or symbolic constants of any type acceptable to the intrinsic function. If a dimension bound expression is of type boolean, the value of the expression is converted to integer; that is, the value is INT(expression).

There are five types of arrays. Each type is distinguished by how it is declared. The five types are:

Fixed-size arrays

A fixed-size array is declared when its dimension bound expression contains only constants or symbolic constants.

Adjustable arrays

An adjustable array is declared when at least one dimension bound expression contains one or more variables of type integer (any size) or type byte. Adjustable arrays can only be used as a dummy argument in a subroutine or function.

Assumed-size arrays

An assumed-size array is declared when its upper bound is represented as an asterisk. Assumed-size arrays can only be used as a dummy argument in a subroutine or function.

Assumed-shape arrays

An assumed-shape array is declared when all dimensions specify no upper bound. Assumed-shape arrays can only be used as a dummy argument in a subroutine or function.

Allocatable arrays

An allocatable array is declared if both the upper and lower bounds for all dimensions are omitted (but not the colon) and the array is not a dummy argument. array. An allocatable array is an array for which storage is allocated at execution time, rather than at compile time.

For more information about arrays as dummy arguments, see Procedure Communication in chapter 8, Program Units.

For more information about allocatable arrays, see Declaring Allocatable Arrays in this chapter.

An array declaration must not reference itself in any of the dimension bound expressions. For example, the following statement is incorrect:

```
REAL A(10, SIZE(A))
```

Also, dimension bound expressions must not reference another entity defined in the same statement. For example, the following statements are incorrect:

```
REAL B(100), A(10, SIZE(B))
REAL A(10, SIZE(B)), B(100)
```

For Better Performance

Adjustable arrays increase execution time; replace with fixed-size arrays whenever possible.

The following examples show declarations and storage patterns for a one-dimensional, a two-dimensional, and a three-dimensional array. Arithmetic values are shown for the array elements, but an array can be of any data type or length .

Array elements are stored in sequential order by column.

Array Declaration:

```
DIMENSION RX(0:5)
```

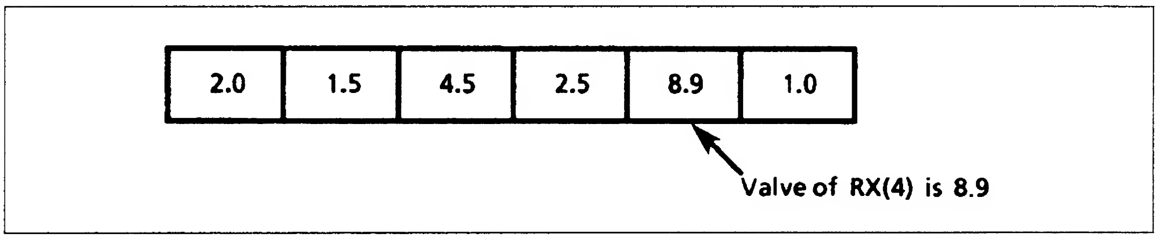


Figure 3-1. Storage Pattern of a One-Dimensional Array

Array Declaration:

```
DIMENSION LAMBDA_1(4, 3)
```

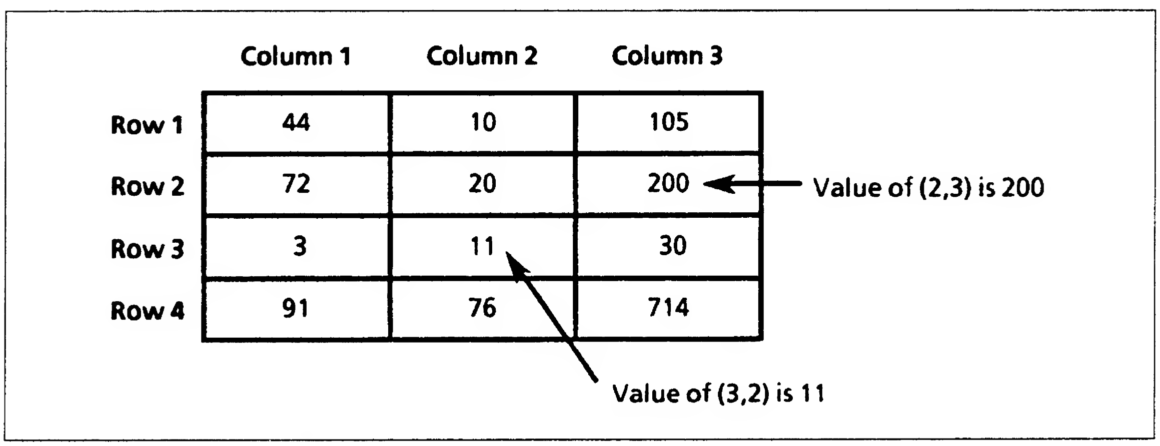


Figure 3-2. Storage Pattern of a Two-Dimensional Array

The array has 4 rows and 3 columns, for a total of 12 elements.

Array declaration:

```
DIMENSION BETA(3, 3, 3)
```

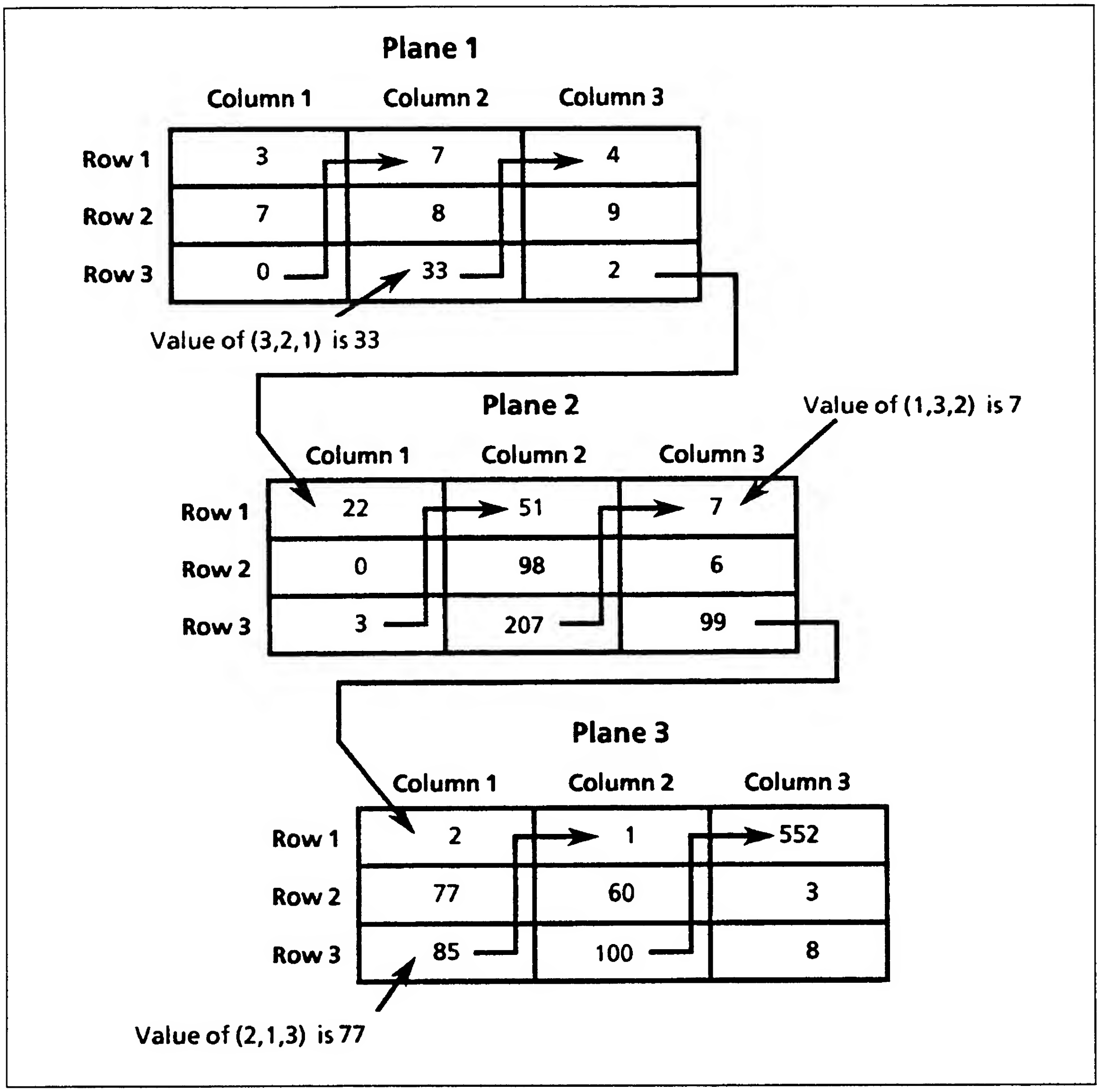


Figure 3-3. Storage Pattern of a Three-Dimensional Array

The array has 3 rows, 3 columns, and 3 planes, for a total of 27 elements.

Control Data Extension

Declaring Allocatable Arrays

Allocatable arrays are arrays for which storage is allocated at runtime, rather than at compile time. Although allocatable arrays must be declared like other arrays, their storage is allocated when the ALLOCATE statement executes and deallocated when the DEALLOCATE statement executes.

ALLOCATE Statement

The ALLOCATE statement allocates storage for an allocatable array. The ALLOCATE statement has the form:

ALLOCATE(name, ..., *name*)

name

A constant or adjustable array name and dimension bound expressions. The array name must have been previously declared as allocatable in a type, COMMON, or DIMENSION statement.

The dimension bound expressions are subject to the same rules as in an array declaration, except that array element references are permitted. For example:

```
DIMENSION A(:,:), J(5)
  :
J(2)=5
ALLOCATE (A(J(2),6))
```

The ALLOCATE statement allocates array A to a 30 element array with 5 rows and 6 columns.

The dimension bound expressions are evaluated when the ALLOCATE statement is executed. The values of the dimension bound expressions determine the sizes of the corresponding dimensions for the allocatable array and the lower and upper bounds of the dimensions. The number of dimensions specified in **name** must be equal to the number specified in the array declaration.

You cannot reference an allocatable array if it is not currently allocated. Also, you cannot reference an array in an ALLOCATE statement in an expression within the same ALLOCATE statement. An ALLOCATE statement cannot reference a currently allocated array.

After successful execution of an ALLOCATE statement for an array, the properties of dimension, size, and lower and upper dimension bounds are established. Any entities in the dimension bounds expressions can be changed with no effect on the above-mentioned properties. Immediately after an array is allocated, the values of all of the array elements become undefined.

An ALLOCATE statement can appear in a main program, subroutine, or function subprogram.

Example:

```
DIMENSION A(:)
      :
ALLOCATE (A(100))
```

The ALLOCATE statement sets the size of A to 100. The lower dimension bound is 1 and the upper dimension bound is 100.

Control Data Extension ***(Continued)***

DEALLOCATE Statement

The DEALLOCATE statement releases the storage for allocated arrays. A DEALLOCATE statement has the form:

DEALLOCATE(name, ..., *name*)

name

Name of an allocatable array previously allocated by execution of an ALLOCATE statement. Results are undefined if **name** is not currently allocated.

Example:

```
DIMENSION TOTAL_COUNT(:)
      :
ALLOCATE(TOTAL_COUNT(200))
      :
DEALLOCATE(TOTAL_COUNT)
```

The array TOTAL_COUNT is allocated and deallocated at runtime.

For Better Performance

Use allocatable arrays only when necessary; they often increase execution time.

End of Control Data Extension

Array Shape and Conformability

The shape of an array or array section is determined by the number of dimensions and the size of each dimension. The number of dimensions is specified when the array is declared. The size of each dimension is the value MAX(*upper-lower+1, 0)* where *upper* is the value of the upper dimension bound and *lower* is the size of the lower dimension bound.

The shape of an array is represented by the size of each dimension enclosed in parentheses, that is, *(d1, d2, ..., dn)* where *n* is the number of dimensions and *d* is the size of the dimension.

For example:

`DIMENSION BIG(5:300,-1:4,50)`	The shape of array BIG is (296,6,50).

Two array objects are of the same shape if they have the same number of dimensions and the sizes of each pair of corresponding dimensions are the same. An array object is an array, array section, array-valued expression, or an intrinsic function reference with an array result.

Two array objects are conformable if they have the same shape. For example:

`DIMENSION A(4:5,3)` `REAL B(2,1:3)`	Arrays A and B have the same shape and are therefore conformable. Both arrays have the shape (2,3).

Array conformability is necessary when using array assignment statements, WHERE statements, and some intrinsic functions. A scalar is always conformable with any array object. The scalar is treated as if it had been extended to an array with the same shape in which all array elements have the value of the scalar.

A runtime check for array conformability can be selected by the RUNTIME_CHECKS parameter on the VECTOR_FORTRAN command.

Array Storage

The elements of an array have a specific storage order, with elements of any array stored as a linear sequence of storage words. Each storage word, or computer word, is 8 bytes. The first element of the array begins at the first storage position and the last element ends at the last storage word or character storage position. An array must not exceed (2**31)-1 bytes in size.

The number of words used for an array is determined by the data type of the array, the length of the data type, and the sizes of its dimensions. The following table shows the storage required by various types and lengths of arrays:

Data Type	Length	Computer Words
Integer	8 bytes	n
Integer	4 bytes	n/2
Integer	2 bytes	n/4
Byte	1 bytes	n/8
Boolean	8 bytes	n
Real	8 bytes	n
Real	16 bytes	2*n
Logical	8 bytes	n
Logical	4 bytes	n/2
Logical	2 bytes	n/4
Logical	1 bytes	n/8
Character	length	n*length/8
Complex	16 bytes	2*n
Double Precision	16 bytes	2*n

where *length* is the length in characters of the array element and *n* is the product of the sizes of all dimensions

NOTE

For a program compiled with OPTIMIZATION_LEVEL=HIGH on the VECTOR_FORTRAN command, storage is not allocated at runtime for arrays unless they are one of the following:

In a common block
Saved (by a SAVE statement or FORCED_SAVE=ON compiler option)
Initialized in a DATA statement
Used as actual arguments

Instead, storage is allocated for them on the runtime stack during execution when the containing program unit becomes active. This storage is then given up on execution of a RETURN or END statement in the program unit.

The default runtime stack size is about 2 million bytes. If this limit is exceeded, a runtime error results, usually of the form *Tried to read/write beyond maximum segment length* or *A stack segment contains invalid frames.*

For programs where the number of active items allocated on the runtime stack exceeds the default limit, you can increase the runtime stack size by specifying the STACK_SIZE parameter on the EXECUTE_TASK command (as described in the NOS/VE Object Code Management manual).

Array References

Array references can be references to complete arrays, specific array elements, or to array sections.

Complete Arrays

You can reference a complete array by specifying the array name with no subscript. A reference to the complete array references all elements of the array in columnwise order. For example:

```
DIMENSION XT(3)
DATA XT / 1., 2., 3. /
PRINT*, XT
XT = XT * 100.
```

The DATA statement, the PRINT statement, and the assignment statement reference all three elements of array XT.

You can specify a complete array, that is, an array name with no subscripts, in the following:

Assignment statements

COMMON statements

DATA statements

EQUIVALENCE statements

NAMELIST statements

SAVE statements

Input/Output statements

Type declaration statements

WHERE statements

ALLOCATE and DEALLOCATE statements

A reference to a complete assumed-size array is permitted only as an actual argument to a subprogram.

Array Elements

To reference a specific element in an array, specify the array name followed by subscripts. A reference to an array element has the form:

array(e, ..., *e*)

array

Array name.

e

Subscript expression that is an integer (any length), byte, real, double precision, complex, or boolean expression.

When referencing an array element, you must specify a subscript expression for each dimension in the array. Each subscript expression is evaluated and converted as necessary to integer. The conversion is done using INT(**e**).

Example:

`B=A(1, 3, 6)`	This references the element in the first row, third column, and sixth plane of A.

A subscript expression can contain function references and array element references; however, evaluation of a function reference must not alter the value of any other subscript expression in the array element reference. (The compiler does not diagnose this condition, however.) The function evaluation also must not alter the value of any other entity in the same subscript expression.

For example:

`B=ARY(2, IFUNC(J) + K)`	In this statement, IFUNC must not alter the value of K.

Each subscript value after conversion to integer must not be less than the lower bound or greater than the upper bound of the dimension. If the array is an assumed-size array, the value of the subscript expression must not exceed the actual size of the dimension.

For example:

```
DIMENSION AR(100)
      :
CALL SUB(AR)

SUBROUTINE SUB(BB)
DIMENSION BB(*)
      :
BB(I + J - K)=0.0
```

In the assignment statement, the value of I + J - K must not exceed 100.

If a subscript expression value falls outside the dimension bounds, the results are undefined. A runtime check for range violations can be selected by the RUNTIME_CHECKS parameter on the VECTOR_FORTRAN command. For each array element reference, evaluation of the subscript expressions yields a value for each dimension and, ultimately, a position relative to the beginning of the array.

The position of an array element is calculated as shown in the following table:

Dimensions in the Array	Position of Array Element
1	1 + (s1 - j1)
2	1 + (s1 - j1) + (s2 - j2) * n1
3	1 + (s1 - j1) + (s2 - j2) * n1 + (s3 - j3) * n2 * n1
⋮	⋮
7	1 + (s1 - j1) + (s2 - j2) * n1 + (s3 - j3) * n2 * n1 + (s4 - j4) * n3 * n2 * n1 + (s5 - j5) * n4 * n3 * n2 * n1 + (s6 - j6) * n5 * n4 * n3 * n2 * n1 + (s7 - j7) * n6 * n5 * n4 * n3 * n2 * n1

where *si* is the value of the subscript expression specified for dimension i.

where *ji* is the lower bound of dimension i.

where *ni* is the size of dimension i; ni = ki - ji + 1; ki is the upper bound of dimension i. If the lower bound is one, ni = ki.

The position of an array element in the table indicates the relative position of an array element.

Example:

```
INTEGER DZ(12)
      :
DZ(6) = 79
```

The array element reference DZ(6) refers to the element at position 6 in the array, that is, position (1 + (6 - 1)).

Example:

```
COMMON /CHAR/ CQ
CHARACTER*3 CQ(6, 4)
      :
CQ(6, 3) = 'RUN'
```

The array element reference CQ(6, 3) refers to the element at position 18, that is, position (1 + (6 - 1) + (3 - 1) * 6). The character storage location is 52 relative to the first position of the array, that is, 1 + (element position - 1) * character length.

For Better Performance

The number of different variable names in subscript expressions should be minimized. For example, the following statements use two variables in each subscript expression:

```
IP1 = I + 1
IM1 = I - 1
X = A(IP1, IM1) + B(IM1, IP1)
```

The following statement produces the same result but executes faster:

```
X = A(I + 1, I - 1) + B(I - 1, I + 1)
```

Control Data Extension

Array Sections

An array section is a sequence of elements in an array. An array section can contain one element, several elements, or all elements of an array. An array section reference has the form:

array(s, ..., *s*)

array

Array name.

s

A section selector or subscript expression. A section selector has the following form:

s1 **:** *s2* *:s3*

where:

s1

A subscript expression for the lower dimension bound of the section. If omitted, the corresponding lower dimension bound of the array is used.

s2

A subscript expression for the upper dimension bound. Must be present if array is assumed size and the section selector is for the last dimension. Otherwise, it can be omitted; in which case the corresponding upper dimension bound of the array is used.

s3

Increment used to select array section elements. If omitted, the increment value is 1.

The subscript expression must be an integer (any length), byte, real, double precision, complex, or boolean expression.

The section selector identifies elements for the dimension position where it is written; the number of dimensions of the array section is equal to the number of section selectors. An array section reference must contain at least one section selector. The order of dimensions in an array section is determined from left to right by the appearance of section selectors.

A subscript expression identifies elements for the dimension position where it is written, and the value of the subscript expression determines the elements to be included in the array section.

Control Data Extension *(Continued)*

For example, given the array declaration

```
DIMENSION A(5, 5, 5)
```

the array section reference A(:, 4, :) is an array section reference with two section selectors and one subscript expression. The subscript expression indicates the fourth element of the second dimension (columns). The first dimension corresponds to the first dimension of A and the second dimension corresponds to the third dimension of A. Thus, the array section consists of all elements (all rows) of the fourth column of all planes.

A reference to an array section references all elements of the array section. Each element selected must be within the bounds of the array being sectioned.

The size of each dimension is the value MAX(*upper-lower+1, 0)* where *upper* is the value of the upper dimension bound and *lower* is the size of the lower dimension bound. The size of each dimension that is referenced by section selectors of the form s1:s2:s3 is MAX(INT((s2 - s1 + s3) / s3), 0). If the total number of elements is positive (s3 cannot equal 0), then the section selector identifies the elements from s1 to s2 in increments of s3. If the upper bound is less than the lower bound, the size of that dimension is zero, and the size of the array section is zero.

The size of an array section is equal to the product of the sizes of the dimensions of the array section.

Example:

```
DIMENSION JAY(4, 4)
```

Array JAY is a two-dimensional array, with each dimension being of size 4. The following diagram represents the integer array JAY and the values it contains:

	Column 1	Column 2	Column 3	Column 4
Row 1	87	29	544	2
Row 2	138	200	68	1023
Row 3	296	45	2	67
Row 4	107	987	90	31

Control Data Extension *(Continued)*

The shading in the following diagram represents the elements of JAY in the array section JAY(1:4, 2):

	Column 1	Column 2	Column 3	Column 4
Row 1	87	29	544	2
Row 2	138	200	68	1023
Row 3	296	45	2	67
Row 4	107	987	90	31

The shading in the following diagram represents elements of JAY in the array section JAY(1:2, 3:4):

	Column 1	Column 2	Column 3	Column 4
Row 1	87	29	544	2
Row 2	138	200	68	1023
Row 3	296	45	2	67
Row 4	107	987	90	31

The shading in the following diagram represents the array section JAY(:, :), which is every element in array JAY:

	Column 1	Column 2	Column 3	Column 4
Row 1	87	29	544	2
Row 2	138	200	68	1023
Row 3	296	45	2	67
Row 4	107	987	90	31

Control Data Extension *(Continued)*

The shading in the following diagram represents the array section JAY(1:4:3, :):

	Column 1	Column 2	Column 3	Column 4
Row 1	87	29	544	2
Row 2	138	200	68	1023
Row 3	296	45	2	67
Row 4	107	987	90	31

End of Control Data Extension

Substrings in Arrays

If a substring reference is used to select a substring from an array element of a character array, the combined reference includes specification of the array element or array section followed by specification of the substring.

For example:

```
CHARACTER*8 ZS(5)
CHARACTER*4 RSEN
      :
ZS(4)(5:6) = 'FG'
RSEN = ZS(1)(:4)
```

The first reference refers to characters 5 and 6 in element 4 of array ZS. The second reference refers to the first four characters of the first element of array ZS.

To assign values to a substring of every element of an array, specify an array section of the entire array. For example:

```
CHARACTER*8 C(10)
C(:) (1:3) = 'ABC'
```

The assignment statement assigns the string 'ABC' to the entire eight characters of the first three elements of C.

Example:

```
CHARACTER*3  A(10,5)
  :
A(1:4,:)(1:2)='RE'
```

The assignment statement assigns the character string 'RE' to the first two character positions of every element in the array section A(1:4,:), which is all columns of the first four rows.

Specification Statements 4

BOOLEAN Statement 4-3
  Boolean Variables 4-3

BYTE Statement 4-4
  Byte Variables 4-4

CHARACTER Statement 4-5
  Character Variables 4-6
    Character Substrings 4-6

COMMON Statement 4-8

COMPLEX Statement 4-11
  Complex Variables 4-11

DATA Statement 4-12
  Implied DO List in a DATA Statement 4-15
  Character Data in a DATA Statement 4-17

DIMENSION Statement 4-18

DOUBLE PRECISION Statement 4-20
  Double Precision Variables 4-20

EQUIVALENCE Statement 4-21
  Equivalence Classes and Common Blocks 4-23

EXTERNAL Statement 4-24

IMPLICIT Statement 4-26

INTEGER Statement 4-28
  Integer Variables 4-29

INTERFACE and END INTERFACE Statements 4-30

INTRINSIC Statement 4-32

LOGICAL Statement 4-34
  Logical Variables 4-34

PARAMETER Statement 4-35

REAL Statement 4-37
  Real Variables 4-37

SAVE Statement 4-38

Specification Statements 4

Specification statements are nonexecutable statements that specify the characteristics of symbolic names used in a FORTRAN program. Specification statements must appear before all DATA statements, NAMELIST statements, statement function statements, and executable statements in the program unit.

DATA statements are not specification statements but are described in this chapter. DATA statements assign initial values to symbolic names.

The specification statements and their purposes are:

Type statements

Each variable, array, symbolic constant, statement function, or external function name has a type. Those entities can be explicitly typed as integer, real, double precision, complex, boolean, byte, logical, or character with the explicit type statements in this chapter. An explicit type statement can also include dimension information for an array. The type statements described in this chapter are BOOLEAN, BYTE, CHARACTER, DOUBLE PRECISION, INTEGER, LOGICAL and REAL.

COMMON

The COMMON statement provides for the sharing of storage. The COMMON statement defines blocks of storage to be shared by multiple program units.

DIMENSION

The DIMENSION statement specifies the number of dimensions in an array and the bounds for each dimension.

EQUIVALENCE

The EQUIVALENCE statement also provides for the sharing of storage. The EQUIVALENCE statement allows variables within a program unit to share storage locations.

EXTERNAL

The EXTERNAL statement controls the recognition of function names. The EXTERNAL statement specifies that a function name refers to a user-written function rather than an intrinsic function or a variable (in an actual argument list).

IMPLICIT and IMPLICIT NONE

The IMPLICIT statement specifies the data types of objects through their symbolic names. Default typing of names takes place unless the type statements are used to change the data type of specific names. The IMPLICIT statement changes the default typing of names. Any IMPLICIT statements or IMPLICIT NONE statements must precede all other specification or DATA statements, except PARAMETER statements. The IMPLICIT NONE statement specifies that there is no implied typing for symbolic names in a program unit.

INTERFACE and END INTERFACE

The INTERFACE and END INTERFACE statements are used when a subroutine or function contains an assumed-shape array as a dummy argument.

INTRINSIC

The INTRINSIC statement also controls the recognition of function names. The INTRINSIC statement specifies that a function name refers to an intrinsic function (supplied by the compiler or a runtime library) rather than a user-written function.

PARAMETER

The PARAMETER statement gives a symbolic name to a constant. PARAMETER statements can appear anywhere among the specification statements. However, each symbolic constant must be defined in a PARAMETER statement before the first reference to the symbolic constant.

SAVE

The SAVE statement preserves the values of variables after execution of a RETURN or END statement in a subprogram. Variables that would otherwise become undefined remain defined and can be used in any subsequent executions of the same subprogram.

DATA statements give initial values to variables. DATA statements must appear after all specification statements in the program unit.

NOTE

A variable is considered undefined until a value is assigned to it by a DATA statement, input statement, or assignment statement. You should always define a variable before the first reference to the variable. Use of undefined variables in expressions can cause runtime errors or unpredictable results.

The specification statements are described in alphabetical order.

Control Data Extension

BOOLEAN Statement

The BOOLEAN statement declares a name to be of type boolean. The BOOLEAN statement has the form:

BOOLEAN name,...,*name*

name

A name that is explicitly typed as boolean. Each name has one of the forms:

var

A variable, symbolic constant, statement function, or external function name.

array(*d*,...,*d*)

An array name with optional dimension bounds specification.

Boolean Variables

A boolean variable is a variable that is typed as boolean. A boolean variable occupies one storage word. Boolean string, octal, or hexadecimal values are usually assigned to boolean variables.

Example:

`BOOLEAN HVAL, ZZZ, R34`	The variables HVAL, ZZZ, and R34 are declared boolean.

End of Control Data Extension

Control Data Extension

BYTE Statement

The BYTE statement declares a name to be of type byte. A byte entity is a 1-byte integer value. The BYTE statement has the form:

BYTE name,...,*name*

name

A name that is explicitly typed as byte. Each name has one of the forms:

var

A variable, symbolic constant, statement function, or external function name.

array(*d,...,d*)

An array name with optional dimension bounds specification.

Byte Variables

A byte variable is typed explicitly as byte. A byte variable contains an integer value that is 1 byte long. The byte data type is intended to provide compatibility with other versions of FORTRAN.

Example:

`BYTE  BIT, FLAG1`	BIT and FLAG1 are declared type byte.

End of Control Data Extension

CHARACTER Statement

The CHARACTER statement declares a name to be of type character. The CHARACTER statement has the form:

CHARACTER*len* **name**,...,*name*

name

A name that is explicitly typed as character. Each name has one of the forms:

var*len*
array*(d,...,d)*len*

where:

var

A variable, symbolic constant, statement function, or external function name.

array*(d,...,d)*

An array name with optional dimension bounds specification.

**len*

Specifies the length (number of characters) of the entity. The length can be one of the following:

An unsigned nonzero integer constant

An extended integer constant expression, enclosed in parentheses, with a positive nonzero value

An asterisk enclosed in parentheses.

If omitted, the length is one character.

A length specification immediately following the keyword CHARACTER applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification for that entity only. For an array, the length specification is for every array element.

If the length specification is a symbolic constant, you must enclose it in parentheses.

Example:

```
CHARACTER A*3, B(10)*(12 + 3*2)
```

The example defines a character variable A that is 3 characters long and a character array B that has 10 elements, each of which is 18 characters long.

You can specify the length of a character dummy argument in an external function or subroutine as (*). A character string with length (*) is called an assumed-length character string.

You can specify the length of a character external function in a FUNCTION or ENTRY statement as (*). When the function is executed, the function has the length specified in the referencing program unit. If the character external function specifies a length other than (*), it must agree with the length specified for the function in the program unit that references the function. There is always agreement of length if the function specifies (*) for its length.

For more information on assumed-length character strings and external functions, see Assumed-Length Character Strings in chapter 8, Program Units.

If you specify a length of (*) for a symbolic constant in a character statement, the constant has the length of its corresponding extended character constant expression in the PARAMETER statement.

Character Variables

A character variable is a variable that is typed as character. The maximum length of a value assigned to a character variable is 65,535 characters.

Examples:

```
CHARACTER NAM*15, C3*3
CHARACTER LAST_NAME*40
```

The variable NAM is 15 characters long, the variable C3 is three characters long, and the variable LAST_NAME is 40 characters long.

```
PARAMETER(N = 5)
CHARACTER*(N) AB, CD*(N + 3)
```

The variable AB is 5 characters long, and the variable CD is 8 characters long.

```
CHARACTER*10 ASTR, ABC(5), XR*20
```

Both the variable ASTR and each element of the array ABC are 10 characters long. The variable XR is 20 characters long.

```
CHARACTER AR*5, BR*8
      :
CALL ZC(BR)
CALL ZC(AR)
      :
END
SUBROUTINE ZC(STR)
CHARACTER STR *(*)
```

When subroutine ZC is called the first time and variable BR is passed, the variable STR has length 8. When subroutine ZC is called with variable AR passed, the variable STR has length 5. For more information, see Procedure Communication in chapter 8, Program Units.

Character Substrings

A character substring is a reference to parts of a string specified by a character variable or array. A character substring reference has the form:

char (*first* : *last*)

char

Character variable or character array element reference (arrays are described in chapter 3).

first

Integer, byte, real, double precision, complex, or boolean expression specifying the position of the first character of the substring. If omitted, 1 is used.

last

Integer, byte, real, double precision, complex, or boolean expression specifying the position of the last character in the substring. If omitted, the length of the string is used.

The expression specifying the first character position in the substring is evaluated and converted, if necessary, to integer. The expression can contain array element references and function references, but evaluation of a function reference must not alter the value of other entities in the substring reference.

The specification of the last character position in the substring is an expression subject to the same rules as the specification of first. If **last** is omitted, the value is the length of the string; all characters from the specified first position to the end of the string are included in the substring. The length of a character substring is last - first + 1. The values of first and last are restricted as follows:

1 ≤ first ≤ length

1 ≤ last ≤ length

If last < first, then a zero length substring is created.

Example:

```
CHARACTER*6 S1, S2
DATA S1/'PEARLS'/
S2=S1
```

S1 is a reference to the full string 'PEARLS'. Substring references to the string S1 could be any of the following:

Substring Reference	**Value**
S1(1:3)	'PEA'
S1(3:4)	'AR'
S1(4:)	'RLS'
S1(:4)	'PEAR'
S1(:)	'PEARLS'

The substring reference S1(:) has the same effect as the reference S1 because all characters in the string are referenced. Note that the colon is required in substring references; for example, S1(3) is not a valid substring reference.

If a substring expression exceeds the bounds declared in the CHARACTER statement the results are undefined. The RUNTIME_CHECKS parameter on the VECTOR_FORTRAN command provides a runtime check on substring bounds violations.

Substrings in character arrays are described in chapter 3, Arrays.

COMMON Statement

The COMMON statement defines areas of storage to be available to the program unit in which the statement appears and associates variables and arrays with the defined area of storage. The areas of storage declared in a COMMON statement are called common blocks. The COMMON statement has the form:

COMMON */name/***nlist**, ..., */name/nlist*

name

Name identifying a named common block; if you omit *name*, the common block is called blank (unlabeled) common. If the first specification is for blank common, you can also omit the slashes. The comma separating */name/* from the preceding **nlist** is optional.

nlist

Required; list of entities to be included in the common block. The entities are separated by commas and have one of the forms:

var

Variable name

array*(d,...,d)*

Array name with optional dimension bound specifications.

COMMON blocks associate entities in different program units. The use of common blocks enables different program units to define and reference the same data without using arguments and to share storage locations. Within a program unit, an entity in a common block is known by a specific name. Within another program unit, the same data can be known by a different name.

A particular variable name or array name can appear only once in any COMMON statement within a program unit. You cannot include function or entry names in COMMON statements. In a subprogram, names of dummy arguments cannot be included in COMMON statements.

Variables and arrays in a common block can be of mixed data types. You can use a common block name within a program unit as a variable or array name.

Within a program unit, declarations of common blocks are cumulative. The **nlist** following each successive appearance of *name* (or no name for blank common) adds more entities to that common block and is treated as a continuation of the specification. Variables and arrays are stored in the order in which they appear in **nlist**.

You should ensure that entities in a common block definition are word-aligned if you intend to equivalence the entities. A word contains 8 bytes or characters. See the EQUIVALENCE statement in this chapter for more information.

The maximum number of common blocks in an executable program, including blank and named common, is 500. The maximum size of each common block is 536,870,912 storage words (4,294,967,296 characters or bytes).

The actual size of any common block is the number of storage words required for the entities in the common block, plus any extensions associated with the common block by EQUIVALENCE statements or C$ EXTEND directives. You can make extensions by adding storage words at the end of the common block or declaring the block extensible. (See the EQUIVALENCE statement in this chapter or the C$ EXTEND directive in chapter 11, C$ Directives.)

You can treat a blank common block as having a different size in separate program units. The length of a common block, other than blank common, must not be increased by a subprogram using the block unless the subprogram is loaded first or the common block is extensible. If a program unit does not use all locations reserved in a common block, unused variables can be inserted in the COMMON declaration to ensure proper correspondence of common areas between program units.

Variables and arrays in named common blocks can be initially defined by a DATA statement in a block data subprogram or by a DATA statement in any program unit. Entities in blank common cannot be initialized. After an entity in a named common block has been initialized, the value is available to any subprogram in which the named common block appears. If the entity is reinitialized in a later subprogram, a loader diagnostic is issued.

Variables and arrays in common (blank or named) remain defined at all times and do not become undefined on returning from a subprogram.

Example:

```
COMMON A, B
COMMON /XT/ C, D, E
       ⋮
SUBROUTINE P(Q, R)
COMMON /XT/ F, G, H
       ⋮
FUNCTION T(U)
COMMON Y, Z
```

The entities C, D, and E in the main program are in the common block named XT. The same storage words are known by the names F, G, and H in subroutine P. The entities A and B in the main program are in blank common. The same storage words are known by the names Y and Z in function T.

Example:

```
COMMON JCOUNT
       ⋮
JCOUNT = 6
       ⋮

FUNCTION AB(A)
COMMON /C/ STX(4)
DATA STX/1., 2., 2., 1./
```

Because an entity in blank common cannot appear in a DATA statement, an assignment statement must be used to define the value of JCOUNT. In function AB, a DATA statement can be used to define initial values for the elements of array STX in the common block named C. Note that JCOUNT is not common to function AB.

Example:

```
CHARACTER*4 D, E
INTEGER J
COMMON /CVAL/ D, E, J
DATA D,E,J/'TEST','PROD',1/
```

The common block named CVAL contains two character variables and one integer variable. The variables are initially defined in a DATA statement.

Example:

```
COMMON /SUM/ A, B(20)
      :
SUBROUTINE GR
COMPLEX FR(10)
COMMON /SUM/ X, FR
```

The common block SUM in the main program is declared to contain the variable A and the array B. In the subroutine GR, the same storage words are associated with X and the array FR. Even if X is not used in the subroutine, X holds a place so array FR matches the placement of array B. Note also that array FR is complex. The elements B(1) and B(2) are known in GR as FR(1); B(3) and B(4) are FR(2); and so forth. Common block SUM requires 21 storage words.

For Better Performance

You can access large blocks of data more efficiently using segment access files mapped to common blocks. See the description of C$ SEGFILE in chapter 11, C$ Directives.

COMPLEX Statement

The COMPLEX statement declares a name to be of type complex. The COMPLEX statement has the form:

COMPLEX*len* **name***len*, ..., *name*len*

name

A name that is explicitly typed as complex. Each name has one of the forms:

var

A variable, symbolic constant, statement function, or external function name.

array*(d,...,d)*

An array name with optional dimension bounds specification.

len [1]

Length (in bytes) of name; 16 is the only option. If omitted, the default length is 16 bytes.

A length specification immediately following the COMPLEX keyword applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. For an array, the length specification is for every array element. If a length is not specified for an entity, either explicitly or by the IMPLICIT statement, the length is 16.

Complex Variables

A complex variable is a variable that is typed as complex. A complex variable occupies two storage words; each word contains a real number. The first word contains the real part of the complex number and the second word contains the imaginary part.

Example:

`COMPLEX ZETA, MU, LAMBDA`	The variables ZETA, MU, and LAMBDA are declared complex.

1. The length specification of the COMPLEX statement is provided for compatibility with other versions of FORTRAN.

DATA Statement

The DATA statement provides initial values for variables, arrays, array elements, and substrings. The DATA statement has the form:

DATA nlist/clist/,...,*nlist/clist/*

nlist

A list of names to be initially defined. Each name is one of the following:

A variable name
An array name
An array element name
A character substring name
An implied DO list

clist

A list of constants or symbolic constants, separated by commas, specifying the initial values. Each item in the list has one of the forms:

c
r*c
r(c,...,c)

where

c

is a constant or symbolic constant.

r

A repeat count that is an unsigned nonzero integer constant or symbolic constant. The repeat count repeats the constant or list of constants enclosed in parentheses.

For each **nlist** in the DATA statement, you must specify the same number of items in the corresponding **clist**. A one-to-one correspondence exists between the items specified by **nlist** and the constants specified by **clist**. The first item of **nlist** corresponds to the first constant of **clist**, the second item to the second constant, and so forth. If an unsubscripted array name appears as an item in **nlist**, you must specify a constant in **clist** for each element of the array. The values of the constants are assigned according to the storage order of the array.

For arithmetic data types, the constant is converted to the type of the associated **nlist** item if the types differ. For all other types, the data type of each constant in **clist** must be compatible with the data type of the **nlist** item. The correspondence is shown in the following table:

Data Type of nlist Item	Data Type of Corresponding clist Constant
Integer, byte, real, double precision, complex, or boolean	Integer, byte, real, double precision, complex, character, or boolean. The value of the **nlist** item is the same as would result from an assignment statement of the form: **nlist**-item = **clist**-constant. Only the first word of double precision or complex **nlist** data is defined by boolean or character **clist** data.
Logical	Logical
Character	Character

If the **nlist** entity is of type double precision and the **clist** constant is of type real, the processor may supply more precision derived from the constant than can be contained in a real data item.

If the **nlist** item is an arithmetic type and **clist** item is of type character, the hexadecimal value of the character string is assigned to the arithmetic value, left justified with blank fill. (Blank fill is ASCII code 20 hexadecimal.)

A character constant associated with an arithmetic or boolean data item is treated as a boolean string constant.

Each subscript expression used in an array element name in **nlist** must be an extended integer constant expression, except that implied DO variables can be used if the array element name is in **clist**. A reference to an implied DO variable must be within the range of the implied DO. Each substring expression used for an item in **nlist** must be an extended integer constant expression. See Extended Integer Constant Expressions in chapter 2, Language Elements.

The DATA statement is nonexecutable and can appear anywhere after the specification statements in a program unit. Usually, DATA statements are placed after the specification statements but before the statement function definitions and executable statements.

Entities that are not in a DATA statement (or associated with any entity in a DATA statement) are undefined at the beginning of execution of the program.

If an entity is defined more than once in a DATA statement in the same program unit, the last definition overrides any previous ones.

You must not initialize a variable, array element, or substring more than once in separate program units. If two entities are associated, or equivalenced, only one can be initially defined by a DATA statement.

Names of dummy arguments, functions, and entities in blank, extensible or segmented access common (including any entities associated these entities) cannot be in a DATA statement. Entities in a named common block can be initialized within a block data subprogram, or within any program unit in which the named common block appears.

Example:

```
INTEGER K(6)
DATA JR/4/
DATA AT/5.0/, AQ/7.5/
DATA NRX, SRX/17.0, 5.2/
DATA K/1, 2, 3, 3, 2, 1/
```

The variables JR, AT, AQ, and SRX are initially defined with the values 4, 5.0, 7.5, and 5.2, respectively. The variable NRX is initially defined with the value 17, after type conversion of the real 17.0 to the integer 17. The array K with six elements is initially defined with a value for each array element.

Example:

```
REAL R(10, 10)
DATA R/50 * 5.0, 50 * 75.0/
```

The array R is initially defined with the first 50 elements set to the value 5.0 and the remaining 50 elements set to the value 75.0.

Example:

```
DIMENSION TQ(2)
EQUIVALENCE(RX, TQ(2))
DATA TQ(1)/32.0/
DATA RX/47.5/
```

The first element of array TQ is initially defined with the value 32.0. The variable RX and the second element of array TQ are initially defined as 47.5, since TQ(2) is equivalenced to variable RX.

Example:

```
BOOLEAN MASK
DATA MASK /Z"FFFF"/
```

The variable MASK is initially defined with the hexadecimal value 00...00FFFF.

Implied DO List in a DATA Statement

You can use an implied DO list as an item in **nlist** of a DATA statement. The implied DO list has the form:

(**dlist**, **i** = **init**, **term**, *incr*)

dlist

A list of array element names and implied DO lists. Subscript expressions must consist of extended integer constant expressions and active DO variables from **dlist**, except that the expression can contain implied DO variables of implied DO lists that have the subscript expression within their ranges.

i

An integer variable called the implied DO variable.

init

An extended integer constant expression specifying the initial value of **i**; can contain implied DO variables of other implied DO lists that have this DO list within their ranges or a variable defined in the same DATA statement.

term

An extended integer constant expression specifying the final value for **i**; can contain implied DO variables of other implied DO lists that have this DO list within their ranges or a variable defined in the same DATA statement.

incr

An optional extended integer constant expression specifying the increment for **i**; can contain implied DO variables of other implied DO lists that have this DO list within their ranges. Cannot equal zero. Default value is 1.

An iteration count and the values of the implied DO variable are established from **init**, **term**, and *incr* just as for DO loops, except that the iteration count must be positive.

When the implied DO list appears in a DATA statement, the list items in **clist** are specified once for each iteration of the implied DO list, with appropriate substitution of values for any occurrence of the implied DO variable **i**.

The appearance of a name as an implied DO variable in a DATA statement does not affect the value or definition status of a variable with the same name in the program unit.

Example:

```
REAL X(5, 5)
DATA((X(J, I), I = 1, J), J = 1, 5) /15*1.0/
```

Elements of array X are initially defined with the DATA statement. Elements in the lower diagonal part of the matrix are set to the value 1.0. The elements initialized are:

(1, 1)				
(2, 1)	(2, 2)			
(3, 1)	(3, 2)	(3, 3)		
(4, 1)	(4, 2)	(4, 3)	(4, 4)	
(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)

Example:

```
PARAMETER(PI = 3.14159)
REAL Y(5, 5)
DATA((Y(J + 1, I), J = I + 1, 4), I = 1, 3) /6 * PI/
```

The following array elements are initially defined with the value 3.14159:

(3, 1)		
(4, 1)	(4, 2)	
(5, 1)	(5, 2)	(5, 3)

Elements of an array that are not explicitly defined in a DATA statement remain undefined. For example:

```
DIMENSION RAY(3)
DATA RAY(2) /0./
```

RAY(1) and RAY(3) are undefined.

Character Data in a DATA Statement

For initialization by a DATA statement, a character item in **nlist** must correspond to a character constant in **clist**. The character item becomes initially defined according to the following rules:

- If the length of the character item in **nlist** is greater than the length of the corresponding character constant, the additional character positions in the item are initially defined as spaces.
- If the length of the character item in **nlist** is less than the length of the corresponding character constant, the additional characters in the constant are ignored.

Initial definition of a character item causes definition of all character positions of the item. Each character constant initially defines exactly one character variable, array element, or substring.

Examples:

```
CHARACTER STR1*6, STR2*3
DATA STR1 /'ABCDE'/
DATA STR2 /'FGHJK'/
```

The character variables STR1 and STR2 are initially defined. Variable STR1 is set to 'ABCDE ', with the sixth character position defined as a space. Variable STR2 is set to 'FGH', with the fourth and fifth characters of the constant ignored.

```
CHARACTER STRING*6
DATA STRING(2 : 5)/'BCDEF'/
```

Positions 2 through 5 of STRING are set to BCDE. Positions 1 and 6 of STRING are undefined and the last character of the constant is ignored.

DIMENSION Statement

The DIMENSION statement declares array names and specifies the bounds of each array. The DIMENSION statement has the form:

DIMENSION array(d, ..., *d*), ..., *array(d*, ..., *d)*

array

Array name.

d

Specifies the bounds of a dimension in one of the forms:

lower: **upper**
lower **:**

where:

lower

Specifies the lower bound of the dimension. The lower bound can be an integer, byte, or boolean expression with a positive, zero, or negative value. If *lower* is omitted, the lower bound defaults to 1.

upper

Specifies the upper bound of the dimension. The upper bound can be an integer, byte, or boolean expression with a positive, zero, or negative value. In the case of an assumed-size array, the upper bound of the last dimension must be specified as *.

The number of dimension bounds specified indicates the number of dimensions in the array. Arrays can have one through seven dimensions.

The upper bound is usually greater than or equal to the lower bound. The size of each dimension is the distance between the lower and upper bound, that is, MAX(upper-lower+1,0) where *upper* is the value of the upper dimension bound and *lower* is the value of the lower dimension bound. If the upper bound is less than the lower bound, the size of the dimension is zero and the size of the array is zero.

If both the upper and lower bounds for all dimensions are omitted and the array is not a dummy argument, the array is an allocatable array.

An expression defining a dimension bound can contain symbolic constants defined in previous PARAMETER statements; integer constants can be of any length. In a function or subroutine, the expression can contain dummy arguments.

A dimension bound expression must not include nonintrinsic function references, array section, or array element references. Arguments to intrinsic function references can contain other intrinsic function references, integer variables, constants, or symbolic constants of any type acceptable to the intrinsic function. If a boolean expression is used for the lower or upper bound of a dimension, the value of the expression is converted to integer, that is, the value is INT(expression).

An array declaration appearing in a specification statement must not reference the same array in the dimension bound expression.

An adjustable array is declared with at least one dimension bound expression containing one or more variables of any size integer or data type byte. An assumed-size array is declared with the upper bound represented as an asterisk. An assumed-shape array is declared with all dimensions specifying no upper bound. An adjustable, assumed-size, or assumed-shape array can be used only as a dummy argument in a subroutine or function.

A DIMENSION statement can declare more than one array. Dummy argument arrays specified within a subprogram can have adjustable dimension specifications. For more information on adjustable dimensions, see Procedure Communication in chapter 8, Program Units.

You can declare an array only once within a program unit. You can specify dimension information in COMMON statements or type statements as well as the DIMENSION statement. The dimension information defines the array dimensions and the bounds for each dimension.

The description of arrays in chapter 3 covers the properties of arrays, the storage of arrays, and array references.

Example:

```
REAL NIL
DIMENSION NIL(6, 2, 2)
```

NIL is an array containing 24 real elements. The following statement is equivalent:

```
REAL NIL(6, 2, 2)
```

Example:

```
COMPLEX BETA
DIMENSION BETA(2, 3)
```

BETA is an array containing six complex elements.

Example:

```
CHARACTER*8 XR
DIMENSION XR(0 : 4)
```

XR is an array containing five character elements, and each element has a length of eight characters. A reference to the third and fourth characters of the second element would be XR(1) (3 : 4).

DOUBLE PRECISION Statement

The DOUBLE PRECISION statement declares a name to be of type double precision. The DOUBLE PRECISION statement has the form:

DOUBLE PRECISION name,...,*name*

name

A name that is explicitly typed as double precision. Each name has one of the forms:

var

A variable, symbolic constant, statement function, or external function name.

array*(d,...,d)*

An array name with optional dimension bounds specification.

Double Precision Variables

A double precision variable is a variable that is typed as double precision. The range restrictions for double precision variables are the same for real constants with approximately 29 significant digits of precision. Double precision variables occupy two consecutive storage words. The first word contains the more significant part of the number and the second part contains the less significant part.

Example:

`DOUBLE PRECISION OMEGA, X`	The variables OMEGA and X are declared double precision.

For Better Performance

Double precision and 16-byte real constants, variables, arrays, and functions require more execution time because of the extra precision they support (two words).

EQUIVALENCE Statement

The EQUIVALENCE statement specifies the sharing of storage by two or more entities in a program unit. The EQUIVALENCE statement has the form:

EQUIVALENCE(nlist),...,*(nlist)*

nlist

A list of variable names, array names, array element names, or character substring references. The names are separated by commas. Each **nlist** establishes an equivalence class within a program unit. Each subscript or substring expression in the list must be an extended integer constant expression.

Equivalencing causes association of the entities in the equivalence class within a program unit.

If entities in an equivalence class are of different data types, equivalencing does not cause type conversion. (The ANSI Standard does not allow equivalencing of character and non-character data.) If a variable and an array are equivalenced, the variable does not acquire array properties, and the array does not lose the properties of an array. The lengths of equivalenced character entities can be different.

Each **nlist** specification must contain at least two names of entities to be equivalenced. In a subprogram, names of dummy arguments cannot appear in the list. Function and entry names cannot be included in the list. Equivalencing specifies that all entities in the list share the same first storage positions. Equivalencing can indirectly cause the association of other entities, for instance, when an EQUIVALENCE statement interacts with a COMMON statement.

If an array element is included in **nlist**, the number of subscript expressions must match the number of dimensions declared for the array name. If an array name appears without a subscript in the list, the effect is as if the first element of the array had been included in the list.

Example:

```
DIMENSION Y(4), B(3, 2)
EQUIVALENCE(Y(1), B(3, 1))
EQUIVALENCE(X, Y(2))
```

Storage is shared so that six storage words are needed for Y, B, and X. The associations are:

	B(1, 1)	
	B(2, 1)	
Y(1)	B(3, 1)	
Y(2)	B(1, 2)	X
Y(3)	B(2, 2)	
Y(4)	B(3, 2)	

Example:

```
CHARACTER A*5, C*3, D(2)*2
EQUIVALENCE (A, D(1)), (C, D(2))
```

Storage is shared so that five character storage positions are needed for A, C, and D. The associations are:

A(1:1)	D(1)(1:1)	
A(2:2)	D(1)(2:2)	
A(3:3)	D(2)(1:1)	C(1:1)
A(4:4)	D(2)(2:2)	C(2:2)
A(5:5)		C(3:3)

You can equivalence variables of different data types. The equivalencing associates the first storage word of each entity for 8-byte data types. For example,

```
REAL TR(4)
COMPLEX TS(2)
EQUIVALENCE(TR, TS)
```

causes the following associations:

TR(1)	TS(1)-real part
TR(2)	TS(1)-imaginary part
TR(3)	TS(2)-real part
TR(4)	TS(2)-imaginary part

Equivalencing must not reference array elements in a way that conflicts with the storage sequence of the array. You cannot specify the same storage unit as occurring more than once in the storage sequence. For example,

```
REAL FA(3)                          }
EQUIVALENCE(FA(1), B), (FA(3), B)   } Not legal
```

Also, the normal storage sequence of array elements cannot be interrupted to make consecutive storage words no longer consecutive. For example,

```
REAL BZ(7), CZ(5)                      }
EQUIVALENCE(BZ, CZ), (BZ(3), CZ(4))    } Not legal
```

Equivalence Classes and Common Blocks

Equivalencing associates entities within a program unit , while common blocks associate entities across program units. You cannot equivalence entities in common blocks to one another; however, you can equivalence an entity not explicitly declared to be in common to an entity in common without conflict.

The interaction of COMMON and EQUIVALENCE statements is restricted in two ways:

1. An EQUIVALENCE statement must not attempt to associate two different common blocks in the same program unit. For example,

   ```
   COMMON /LT/ A, T     }
   COMMON /LX/ S, R     } Not Legal
   EQUIVALENCE(T, S)    }
   ```

2. An EQUIVALENCE statement must not cause a common block to be extended by adding storage words before the first storage word of the common block. However, a common block can be extended through equivalencing if storage words are added at the end of the common block. For example,

   ```
   COMMON /X/ A         }
   REAL B(5)            } Not legal
   EQUIVALENCE(A, B(4)) }
   ```

   ```
   COMMON /X/ A         }
   REAL B(5)            } Legal
   EQUIVALENCE(A, B(1)) }
   ```

Word-aligned items cannot be equivalenced to byte-aligned common block items unless the word-aligned item is equivalenced on a word boundary. For example:

```
BYTE A(10)
COMMON A
INTEGER I, J, K
EQUIVALENCE (A(2), I)
EQUIVALENCE (A, J)
EQUIVALENCE (A(9), K)
```

The first EQUIVALENCE statement causes incorrect word alignment; the second two EQUIVALENCE statements cause correct word alignment.

EXTERNAL Statement

The EXTERNAL statement identifies a name as representing an external function or subroutine and allows that name to be used as an actual argument. The EXTERNAL statement has the form:

EXTERNAL name,..., *name*

name

Name of an external function or subroutine, dummy function or subroutine, or a block data subprogram.

A name can appear only once in all of the EXTERNAL statements of a program unit. If an external subprogram name is an actual argument in a program unit, it must appear in an EXTERNAL statement in the program unit. A statement function name must not appear in an EXTERNAL statement.

If the name of an intrinsic function appears in an EXTERNAL statement in a program unit, the name becomes the name of an external function or subroutine. The intrinsic function with the same name cannot be referenced in the program unit.

Specifying the name of a block data subprogram in an EXTERNAL statement causes the loader to search the object libraries for the block data subprogram.

Example:

```
SUBROUTINE ARGR
EXTERNAL SQRT
      :
Y = SQRT(X)
      :
END
FUNCTION SQRT(XVAL)
      :
END
```

In this example, the name SQRT is declared as external. The function reference SQRT(X) is therefore taken to reference the user-written function SQRT rather than the intrinsic function SQRT. The user-written function must reside in the same object file or library as the referencing program.

Example:

```
SUBROUTINE CHECK
EXTERNAL LOW, HIGH
      :
CALL AR(LOW, VAL)
      :
CALL AR(HIGH, VAL)
RETURN
END

SUBROUTINE AR(FUNC, VAL)
VAL = FUNC(VAL)
      :
RETURN
END
REAL FUNCTION LOW(X)
      :
END

REAL FUNCTION HIGH(X)
      :
END
```

In this example, the names LOW and HIGH are declared as external. In the first call to subroutine AR, LOW is passed as an actual argument so the function reference FUNC(VAL) is equivalent to LOW(VAL). In the second call to subroutine AR, the function reference FUNC(VAL) is equivalent to HIGH(VAL).

IMPLICIT Statement

The IMPLICIT statement changes or confirms the default typing of names according to the first letter of the names. IMPLICIT NONE specifies that there should be no default implied typing for symbolic names in a program unit. The IMPLICIT statements have the form:

IMPLICIT NONE

IMPLICIT type(ac, ..., *ac)* , ..., *type(ac,* ..., *ac)*

type

One of the keyword values INTEGER**len*, REAL**len*, DOUBLE PRECISION, COMPLEX**len*, BOOLEAN, BYTE, LOGICAL**len*, or CHARACTER**clen*.

ac

A single letter, or a range of letters represented by the first and last letter separated by a hyphen, indicating which variables are implicitly typed.

clen

Specifies the length of all entities of type character. Can be an unsigned nonzero integer constant or a positive extended integer constant expression enclosed in parentheses. If you do not specify *clen*, the length is one.

len [2]

Specifies the length, in bytes, of entities. Must be an extended integer constant expression with one of the following values:

Integer entities: 2, 4, or 8

Real entities: 8 or 16

Logical entities: 1, 2, 4, or 8

Complex entities: 16

The IMPLICIT statement establishes the type of variables, arrays, symbolic constants, statement functions, and external functions that begin with the specified letter or range of letters. Since uppercase and lowercase letters are equivalent, implicitly typing a letter of one case has the same effect for the letter of the other case. Intrinsic functions are not affected by the IMPLICIT or IMPLICIT NONE statement.

The IMPLICIT statements in a program unit must precede all other specification statements except PARAMETER statements. An IMPLICIT statement in a function or subroutine subprogram affects the type associated with dummy arguments and the function name, as well as other variables in the subprogram. Any explicit typing with a type or FUNCTION statement overrides both the default typing and any implicit typing.

---

2. The length specification on the IMPLICIT statement provides compatibility with other versions of FORTRAN.

The specified single letters or ranges of letters specify the entities to be typed. A range of letters has the same effect as writing a list of the single letters within the range. The same letter can appear as a single letter or be within a range of letters only once in all IMPLICIT statements in a program unit.

If a program, subroutine, or function contains an IMPLICIT NONE statement, then all names of variables, arrays, symbolic constants, external functions, and statement functions within that program unit must be explicitly declared in a type statement.

No other form of the IMPLICIT statement can be used with the IMPLICIT NONE statement.

The use of IMPLICIT NONE in a program unit causes the detection of undeclared names.

Examples:

`IMPLICIT CHARACTER*20 (M, X - Z)`	The default typing is changed for names beginning with the letters M, X, Y, or Z. Names beginning with M are typed as character rather than integer, and names beginning with X, Y, or Z are typed as character rather than real.
`IMPLICIT INTEGER*2 (I-L)`	The default typing is changed for names beginning with the letters I, J, K, or L. These names are typed as integers occupying 2 bytes of memory each; their range is -2**15 through (2**15)-1.
`IMPLICIT BYTE(B)`	All names beginning with B are type byte.
`IMPLICIT LOGICAL(L)` `INTEGER L, LATEST_X, TT`	The names L and LATEST_X are explicitly typed as integer. All other names beginning with L are implicitly typed as logical. The name TT is explicitly typed as integer and does not take the default type real.
`IMPLICIT NONE`	The IMPLICIT NONE requires that all variables in the program be explicitly typed using a type statement (BOOLEAN, BYTE, CHARACTER, COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, or REAL).

INTEGER Statement

The INTEGER statement declares a name to be of type integer and of length 2, 4, or 8 bytes. To declare an integer of length 1 byte, use the BYTE statement. The INTEGER statement has the form:

INTEGER**len* **name****len*,...,*name*len*

name

A name that is explicitly typed as integer. Each name has one of the forms:

var

A variable, symbolic constant, statement function, or external function name.

array*(d,...,d)*

An array name with optional dimension bounds specification.

len [3]

Length (in bytes) of name. Options are 2, 4, and 8. If omitted, the length is 8 bytes.

A length specification immediately following the INTEGER keyword applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. For an array, the length specification is for every array element. If a length is not specified for an entity, either explicitly or by the IMPLICIT statement, the length is 8.

3. The length specification of the INTEGER statement is provided for compatibility with other versions of FORTRAN.

Integer Variables

An integer variable is a variable that is typed explicitly, implicitly, or by default as integer.

Byte variables are 1-byte integer variables, and are declared explicitly by the BYTE statement or the IMPLICIT statement. The ranges for integer and byte variables are as follows:

Range	Variable Length
-(2**7) through (2**7)-1	1-byte integer (type byte)
-(2**7)-1 through -(2**15)-1 (2**7) through (2**15)-1	2-byte integer
-(2**15)-1 through -(2**31) (2**15) through (2**31)-1	4-byte integer
-(2**31)-1 through -(2**63) (2**31) through (2**63)-1	8-byte integer

Note:

(2**63)-1 is equal to 9,223,372,036,854,775,807.
(2**31)-1 is equal to 2,147,483,647.
(2**15)-1 is equal to 32,767.
(2**7)-1 is equal to 127.

For Better Performance

8-byte variables and functions execute faster than 4-, 2-, and 1-byte variables and functions. This is because such 4-, 2-, and 1-byte variables are byte-aligned rather than word-aligned and therefore slower to load and store.

Examples:

`INTEGER SUM`	SUM is explicitly declared type integer (8 bytes in length).
`INTEGER ITEM1`	ITEM1 is explicitly declared type integer (8 bytes in length).
`NSUM = 3`	NSUM is declared integer by default (provided it does not appear in a type statement in the same program unit).
`INTEGER*2 SMALL_VALUE`	SMALL_VALUE is explicitly declared type integer of 2 bytes in length.

NOTE

Throughout this manual, whenever an integer constant, variable, or expression is allowed, it can be of any length, or of data type byte, unless otherwise noted.

Control Data Extension

INTERFACE and END INTERFACE Statements

An interface block begins with an INTERFACE statement and ends with an END INTERFACE statement. An interface block provides information about an external function or subroutine. An interface block is required if the function or subroutine contains a dummy argument that is an assumed-shape array. An interface block has the form:

```
INTERFACE
    Interface header
    Specification statements
END INTERFACE
```

The Interface header is either a SUBROUTINE or FUNCTION statement.

The Specification statements can be one or more of the following:

IMPLICIT statement
PARAMETER statement
DIMENSION statement
EXTERNAL statement
Type statements

The specification statements describe the dummy arguments of the function or subroutine specified in the interface header. They must indicate the type and number of all dummy arguments and the array specification of any array dummy arguments in the function or subroutine specified in the interface header. The information contained in the specification statements of the interface block must agree with the information in the specification statements of the function or subroutine.

No other statements can occur within the interface block. An interface block is considered to be within the containing program for C$ directive scoping purposes. For example, if an interface block appears between a C$ LIST (S=0) and a C$ LIST (S=1) directive, the block would not appear on a listing, just as other statements between the directives would not appear.

The interface block identifies to the calling program the interface appearing in the actual subprogram. If the name in the interface header is a function name, the actual type of the function must agree with the type specified in the interface block of the main program.

Control Data Extension *(Continued)*

Example:

Main program:

```
LOGICAL TEST_ARRAY(10, 20)
      :
INTERFACE
  SUBROUTINE NEGATE_VALUES(L1)
  LOGICAL L1(1:, 1:)
END INTERFACE
      :
CALL NEGATE_VALUES(TEST_ARRAY)
```

Interface block for subroutine NEGATE_VALUES

Subroutine:

```
SUBROUTINE NEGATE_VALUES(L1)    --->
LOGICAL L1(1: ,1:)
L1 = .NOT. L1
  :
RETURN
END
```

Assumed-shape dummy argument

End of Control Data Extension

INTRINSIC Statement

The INTRINSIC statement identifies a name as representing an intrinsic function and enables use of an intrinsic function name as an actual argument. The INTRINSIC statement has the form:

INTRINSIC name,...,*name*

name

An intrinsic function name

If you use an intrinsic function name as an actual argument in a program unit, it must appear in an INTRINSIC statement in the program unit. You cannot use the following intrinsic function names as actual arguments:

- Type conversion functions: BOOL, CHAR, CMPLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, and SNGL
- Lexical relationship functions: LGE, LGT, LLE, and LLT
- Largest/smallest value functions: AMAX0, AMAX1, AMIN0, AMIN1, DMAX1, DMIN1, MAX, MAX0, MAX1, MIN, MIN0, and MIN1
- Logical and masking functions: AND, COMPL, EQV, MERGE, NEQV, OR, XOR
- Array-processing functions: ALL, ANY, ALLOCATED, COUNT, DOTPRODUCT, MATMUL, MAXVAL, MINVAL, LBOUND, PACK, PRODUCT, RANK, SEQ, SHAPE, SIZE, SUM, UBOUND, and UNPACK

The appearance of a generic intrinsic function name in an INTRINSIC statement does not remove the generic properties of the name.

An intrinsic function name can appear only once in all INTRINSIC statements in a program unit. A symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

Intrinsic functions that are passed to a subprogram containing the INTRINSIC statement cannot have array-valued arguments within that subprogram.

Example:

```
SUBROUTINE DC
REAL X, Y
INTRINSIC SQRT
      ⋮
CALL SUBA(X, Y, SQRT)
      ⋮
END

SUBROUTINE SUBA(A, B, FNC)
B = FNC(A)
      ⋮
```

The name SQRT is declared intrinsic in subroutine DC and passed as an argument to subroutine SUBA. Within SUBA, the reference FNC(A) references the intrinsic function SQRT.

Example:

```
SUBROUTINE CHECK
REAL VAL
INTRINSIC SIN, COS
      ⋮
CALL AR(SIN, VAL)
      ⋮
CALL AR(COS, VAL)
      ⋮
END

SUBROUTINE AR(FUNC, VAL)
VAL = FUNC(VAL)
      ⋮
```

The names SIN and COS are declared as intrinsic and can therefore be passed as actual arguments. In the first call to subroutine AR, the reference FUNC(VAL) is equivalent to SIN(VAL); in the second call, FUNC(VAL) is equivalent to COS(VAL). In each case, the intrinsic function is referenced.

LOGICAL Statement

The LOGICAL statement declares a name to be of type logical and of length 1, 2, 4 or 8 bytes. The LOGICAL statement has the form:

LOGICAL*len* **name***len*,..., *name*len*

name

A name that is explicitly typed as logical. Each name has one of the forms:

var

A variable, symbolic constant, statement function, or external function name.

array*(d,...,d)*

An array name with optional dimension bounds specification.

len [4]

Length (in bytes) of name. Options are 1, 2, 4, and 8. If omitted, the length is 8 bytes.

A length specification immediately following the LOGICAL keyword applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. For an array, the length specification is for every array element. If a length is not specified for an entity, either explicitly or by the IMPLICIT statement, the length is 8.

Logical Variables

A logical variable is a variable that is typed explicitly or implicitly as logical. Logical variables can only contain the values .true. (representing the logical value true) or .false. (representing the logical value false).

Example:

`LOGICAL L33, PROVIDER_1`	The variables L33, and PROVIDER_1 are declared logical.

4. The length specification for the LOGICAL statement in provided for compatibility with other versions of FORTRAN.

PARAMETER Statement

The PARAMETER statement gives a symbolic name to a constant. This statement has the form:

PARAMETER(name=exp,...,*name=exp*)

name

Name to be defined.

exp

An extended constant expression

If the name of a constant is of type integer, byte, real, double precision, complex, or boolean, the corresponding expression must be an extended arithmetic or boolean constant expression. If the name is of type character or logical, the corresponding expression must be an extended character constant expression or logical constant expression. (Expressions are described in chapter 5.)

Each name is defined as having the value of the expression that appears to the right of the equal sign, according to the rules for assignment. The name then becomes a symbolic constant. Any symbolic constant that appears in **exp** must have been previously defined in the same or a different PARAMETER statement in the program unit.

You can define a symbolic constant only once in a program unit, and the symbolic name can identify only the corresponding constant. You can specify the type of a symbolic constant by an IMPLICIT statement or type statement before the first appearance of the symbolic constant in a PARAMETER statement.

If the length of a symbolic constant is not the default length for that type, the length must be specified in an IMPLICIT statement or type statement before the first appearance of the symbolic constant. The actual length of the constant is then determined by the length of the value defining it in the PARAMETER statement. The length must not be changed by another IMPLICIT statement or by subsequent statements.

Once defined, a symbolic constant can appear in the program unit in the following ways:

- In an expression in any subsequent statement
- In a DATA statement as an initial value or a repeat count
- In a complex constant as the real or imaginary part
- In a C$ directive as a primary in an expression or as a parameter value
- In a dimension bound expression in an array declaration

A symbolic constant cannot appear in a format specification. A character symbolic constant cannot be used in a substring reference.

Example:

```
PARAMETER(ITER=20, START=5)
CHARACTER CC * (*)
PARAMETER(CC='(I4, F10.5)')
      :
DATA COUNT/START/
      :
DO 410 J=1, ITER
      :
READ CC, IX, RX
```

The symbolic constant START assigns an initial value to variable COUNT, the symbolic constant ITER controls the DO loop, and the symbolic constant CC specifies a character constant format specification.

PARAMETER Statement

The PARAMETER statement gives a symbolic name to a constant. This statement has the form:

PARAMETER(name=exp,...,*name=exp*)

name

Name to be defined.

exp

An extended constant expression

If the name of a constant is of type integer, byte, real, double precision, complex, or boolean, the corresponding expression must be an extended arithmetic or boolean constant expression. If the name is of type character or logical, the corresponding expression must be an extended character constant expression or logical constant expression. (Expressions are described in chapter 5.)

Each name is defined as having the value of the expression that appears to the right of the equal sign, according to the rules for assignment. The name then becomes a symbolic constant. Any symbolic constant that appears in **exp** must have been previously defined in the same or a different PARAMETER statement in the program unit.

You can define a symbolic constant only once in a program unit, and the symbolic name can identify only the corresponding constant. You can specify the type of a symbolic constant by an IMPLICIT statement or type statement before the first appearance of the symbolic constant in a PARAMETER statement.

If the length of a symbolic constant is not the default length for that type, the length must be specified in an IMPLICIT statement or type statement before the first appearance of the symbolic constant. The actual length of the constant is then determined by the length of the value defining it in the PARAMETER statement. The length must not be changed by another IMPLICIT statement or by subsequent statements.

Once defined, a symbolic constant can appear in the program unit in the following ways:

- In an expression in any subsequent statement
- In a DATA statement as an initial value or a repeat count
- In a complex constant as the real or imaginary part
- In a C$ directive as a primary in an expression or as a parameter value
- In a dimension bound expression in an array declaration

A symbolic constant cannot appear in a format specification. A character symbolic constant cannot be used in a substring reference.

Example:

```
PARAMETER(ITER=20, START=5)
CHARACTER CC * (*)
PARAMETER(CC='(I4, F10.5)')
       :
DATA COUNT/START/
       :
DO 410 J=1, ITER
       :
READ CC, IX, RX
```

The symbolic constant START assigns an initial value to variable COUNT, the symbolic constant ITER controls the DO loop, and the symbolic constant CC specifies a character constant format specification.

REAL Statement

The REAL statement declares a name to be of type real and of length 8 or 16 bytes. The REAL statement has the form:

REAL*len* **name** **len,...,name*len*

name

A name that is explicitly typed as real. Each name has one of the forms:

var

A variable, symbolic constant, statement function, or external function name.

array*(d,...,d)*

An array name with optional dimension bounds specification.

len [5]

Length (in bytes) of name. Options are 8 and 16. A name declared with length 16 is written and treated as a double precision value. If omitted, the length is 8 bytes.

A length specification immediately following the REAL keyword applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. For an array, the length specification is for every array element. If a length is not specified for an entity, either explicitly or by the IMPLICIT statement, the length is 8.

Real Variables

A real variable is a variable that is typed explicitly, implicitly, or by default as real. A real variable can occupy one computer word (8 bytes) or two consecutive computer words (16 bytes).

Examples:

`REAL  VECTOR, BETA, I`	VECTOR, BETA, and I are explicitly declared real (8 bytes in length)
`TOTAL2 = 3.0`	TOTAL2 is declared real by default (provided it does not appear in any type statement).

5. The length specification for the REAL statement is provided for compatibility with other versions of FORTRAN.

SAVE Statement

The SAVE statement causes values to be saved after the execution of a RETURN or END statement in a subprogram. The SAVE statement has the form:

SAVE *name,...,name*

name

Variable or array name. The same name must not appear more than once. If no names are specified, all variables and arrays are saved for later executions of programs containing the SAVE statement.

You can also use the FORCED_SAVE parameter on the VECTOR_FORTRAN command to save program entities. Selecting the FORCED_SAVE parameter is equivalent to specifying a SAVE statement in every subprogram compiled.

The names of dummy arguments, subroutines, functions, and entities in a common block must not appear in the SAVE statement. You do not have to specify a common block or an entity within a common block in a SAVE statement; storage given to a named or blank common block is not reused during execution of your program.

Execution of a RETURN statement or an END statement within a subprogram causes the entities within the subprogram to become undefined except the following:

- Entities specified by SAVE statements.
- Entities in blank or named common.
- Entities that have been defined in a DATA statement.
- Entities in a subprogram compiled with OL=DEBUG or LOW specified on the FORTRAN command.
- Entities that are associated with saved entities by EQUIVALENCE statements.

NOTE

A SAVE statement has no effect in a main program or in any subprogram compiled with OPTIMIZATION_LEVEL=DEBUG or LOW specified on the VECTOR_FORTRAN command.

Example:

```
PROGRAM MAIN
COMMON /C1/ G, H
CALL XYZ
      :
END

SUBROUTINE XYZ
COMMON A, D, F
COMMON /C1/ GVAL, HVAL
SAVE
DATA JCOUNT /5/
X = 6.5
      :
RETURN
END
```

The SAVE statement in subroutine XYZ has the effect of saving the value of X as 6.5 for any later executions of the subroutine. Saving of certain other values does not depend on the presence of the SAVE statement. The three entities in blank common and the two entities in common block C1 remain defined.

Since JCOUNT is initially defined and not redefined in the subroutine, JCOUNT remains defined for any later executions of the subroutine.

Expressions and Assignment Statements 5

Expressions 5-1
Arithmetic Expressions 5-2
Variable-Length Integer Expressions 5-4
Operator Precedence 5-6
Arithmetic Constant Expression 5-7
Extended Arithmetic Constant Expression 5-8
Type of an Arithmetic Expression 5-9
Character Expressions 5-12
Operator Precedence 5-12
Character Constant Expression 5-13
Extended Character Constant Expression 5-13
Logical Expressions 5-14
Variable-Length Logical Expressions 5-15
Operator Precedence 5-16
Logical Constant Expression 5-16
Extended Logical Constant Expression 5-16
Boolean Expressions 5-17
Operator Precedence 5-18
Boolean Constant Expression 5-18
Extended Boolean Constant Expression 5-18
Data Type of a Boolean Expression 5-18
Relational Expressions 5-19
Arithmetic and Boolean Relational Expressions 5-20
Character Relational Expressions 5-21
General Rules for Expressions 5-22
Array-Valued Expressions 5-24

Assignment Statements 5-25
Arithmetic Assignment Statement 5-25
Character Assignment Statement 5-27
Logical Assignment Statement 5-28
Boolean Assignment Statement 5-29
Multiple Assignment Statement 5-30

Expressions and Assignment Statements 5

Expressions are composed of operands and operators that define an operation to be performed. Expressions are evaluated to a specific value. They are used in assignment statements to assign values to variables, arrays, array sections, substrings, and array elements.

Expressions

You form expressions by combining operators, operands, and parentheses. There are five types of expressions:

- Arithmetic
- Character
- Logical
- Boolean
- Relational

Each type of expression is described separately in this chapter. General rules for expressions are described after these descriptions.

A constant expression is an expression containing only constant operands. An extended constant expression is a constant expression with a reference to an allowable intrinsic function. Allowable intrinsic functions, if any, are listed under each type of expression.

An expression is either a scalar expression or an array expression. A scalar expression is one in which all operands are scalars. A scalar is any one of the following:

Constant of any type
Symbolic constant of any type
Variable of any type
Array element of any type
Character substring
Function reference with a constant, variable, array, array section, array element, or substring as an argument if the result of the function is a scalar

An array expression is one in which one or more operands is an array operand. An array operand is any one of the following:

Array
Array section
Intrinsic function reference whose value is an array

The result of an array expression is an array result. For more information on array expressions, see Array-Valued Expressions in this chapter. Arrays are described in chapter 3.

Arithmetic Expressions

An arithmetic expression performs an arithmetic computation. The value of an arithmetic expression is always a number of type integer, real, complex, boolean, byte, or double precision.

Arithmetic expressions consist of one or more of the following primaries of arithmetic or boolean type, separated by operators and parentheses:

An unsigned constant
A symbolic constant
A variable
An array reference
An array section reference
An array element
A function reference

An arithmetic primary is one whose data type is integer, real, double precision, byte, or complex. A boolean primary is one whose data type is boolean.

The simplest arithmetic expression is one of the primaries listed above; for example, a constant or a variable. You can form more complicated arithmetic expressions using one or more arithmetic or boolean primaries together with one or more arithmetic operators and parentheses.

The arithmetic operators are:

**	Exponentiation
*	Multiplication
/	Division
+	Addition or identity
-	Subtraction or negation

Examples of arithmetic expressions:

```
- 5 + 2
3.78542
(A - B) * F + C / D ** E
J ** I
+ AVAL
C * D / E
A *(- B)
SIN(X)
3.14D3
R "AB" (boolean)
0(1:2, 3:4) (array section)
```

Parentheses used in expressions must be balanced; every left parenthesis must have a corresponding right parenthesis. An expression that appears within another expression is called a subexpression.

Examples of invalid arithmetic expressions:

B * - A	Consecutive operators not permitted.
(F + (X ** Y)	Right parenthesis missing.

NOTE

Any arithmetic operation whose result is not mathematically defined causes an error in the evaluation of the arithmetic expression. Examples of such operations are:

Dividing by zero

Raising a zero-valued primary to a zero-valued or negative-valued power

Raising a negative-valued primary to a real or double precision power

Control Data Extension

Variable-Length Integer Expressions

The length of an integer constant expression containing items of different lengths is the length of the longest item. For example, given the declarations

```
INTEGER  A*2, B*2, C*4, D*4, E*8, F
BOOLEAN  G
BYTE     J
```

the following expression have the lengths shown:

Expression	Length	Reason
A + B	2 bytes	Both A and B are 2-byte integers.
A + 555000	4 bytes	The value of the constant increases the size of the expression because the constant is outside the range of the 2-byte variable A.
B + G	8 bytes	Boolean values are 8 bytes long.
C * D	4 bytes	The result is 4 bytes long since both C and D are 4-byte values; however, arithmetic overflow will occur if the result of C * D is larger than 4 bytes.
A + E	8 bytes	E, the longest integer in the expression, is 8 bytes long.
F * J	8 bytes	F, the longest integer in the expression, is 8 bytes long.
J + A	2 bytes	A, the longest integer in the expression, is 2 bytes long.
J / 2	1 byte	J is of type byte and 2 is within the range for byte constants.
ABS(3)	2 bytes	The constant 3 is within the range for 2-byte integers.

NOTE

Arithmetic overflow resulting from an operation on 2-byte integers is not detected at compile time or runtime.

Control Data Extension *(Continued)*

An integer constant whose size is not explicitly declared is sized by its value, as follows:

Range	Length
-(2**7) through (2**7)-1	1-byte integer (type byte)
-(2**7)-1 through -(2**15)-1 (2**7) through (2**15)-1	2-byte integer
-(2**15)-1 through -(2**31) (2**15) through (2**31)-1	4-byte integer
-(2**31)-1 through -(2**63) (2**31) through (2**63)-1	8-byte integer

Note:

(2**63)-1 is equal to 9,223,372,036,854,775,807.
(2**31)-1 is equal to 2,147,483,647.
(2**15)-1 is equal to 32,767.
(2**7)-1 is equal to 127.

End of Control Data Extension

Operator Precedence

Operator precedence establishes how an arithmetic expression with two or more operators is interpreted. The following precedence among the arithmetic operators determines the order in which the operands are combined:

**	Highest (evaluated first)
* and /	Intermediate
+ and -	Lowest (evaluated last)

Expressions with the highest precedence are evaluated first. For example, in the expression

```
- A ** 2
```

the exponentiation operator (**) has precedence over the negation operator (-). The operands of the exponentiation operator are combined to form an expression used as the operand of the negation operator. The above expression is the same as the expression:

```
- (A ** 2)
```

You can alter the default order of evaluation of subexpressions within an expression by enclosing the subexpression in parentheses. The subexpression within the parentheses is always evaluated before being combined with other operands in the expression. For example, in the expression

```
A + B / C
```

B is divided by C and the result is added to A. However, in the expression

```
(A + B) / C
```

A is added to B and the result is divided by C.

Parenthetical subexpressions can be nested within an expression. For example:

```
(A + (C / D)) * B
```

In an expression containing nested subexpressions, the subexpression within the innermost pair of parentheses is evaluated first.

The direction in which operators of equal precedence are evaluated depends on the operator. Addition, subtraction, multiplication, or division operators are evaluated from left to right. Exponentiation operators are evaluated from right to left. For example:

2 ** 3 ** 2 has the same interpretation as 2 ** (3 ** 2)

5 * 2 / 4 has the same interpretation as (5 * 2) / 4

6 - 7 + 1 has the same interpretation as (6 - 7) + 1

Expression containing two or more consecutive arithmetic operators, such as A ** - B or A + - B, are not permitted. However, expressions such as A ** (- B) and A + (- B) are permitted.

NOTE

The compiler may reorder individual operations in expressions to optimize execution such as the removal of repeated subexpressions. The reordering can cause unexpected results for real, double precision, and complex type data due to truncation errors. Only expressions that are mathematically associative or commutative are candidates for reordering. For example:

Source Expression	Possible Reordering
A / B * C	(A / B) * C (A * C) / B

You can inhibit reordering of operations of equal precedence by using parentheses or specifying EXPRESSION_EVALUATION=CANONICAL on the VECTOR_FORTRAN command.

Arithmetic Constant Expression

An arithmetic constant expression is an arithmetic expression with no variables, array elements, or function references. It contains only boolean or arithmetic constants, boolean or arithmetic symbolic constants, or any of these enclosed in parentheses or preceded by the indentity (+) or negation (-) operator.

If the exponent *e* in an arithmetic constant expression such as *a* ** *e* is of type boolean, the value used is *INT(e)*.

An integer constant expression is an arithmetic constant expression or a boolean constant expression in which each constant or symbolic constant is of type integer or boolean. If a boolean constant expression *e* appears, the value used is *INT(e)*.

Examples of integer constant expressions:

```
3
- 3 + 4
R "A"
- 3
O "74"
R "AB" .AND. 48
```

The length of an integer constant expression is determined by the value of the constants in the expression.

Extended Arithmetic Constant Expression

An extended arithmetic constant expression is an arithmetic constant expression that also contains a reference to any of the following intrinsic functions. Arguments to the intrinsic functions must be arithmetic constant expressions, arrays, or array sections with a scalar result.

ABS	BOOL	DATAN2	DSINH	ISIGN	REAL
ACOS	CABS	DBLE	DSQRT	LBOUND[2]	SHAPE
AIMAG	CCOS	DCOS	DTAN	LEN	SHIFT
AINT	CEXP	DCOSH	DTANH	LOG	SIGN
ALOG	CLOG	DDIM	EQV	LOG10	SIN
ALOG10	CMPLX	DEXP	ERF	MASK	SIND
AMAX0	COMPL	DIM	ERFC	MAX	SINH
AMAX1	CONJG	DINT	EXP	MAX0	SIZE[2]
AMIN0	COS	DLOG	FLOAT	MAX1	SNGL
AMIN1	COSD	DLOG10	IABS	MIN	SQRT
AMOD	COSH	DMAX1	ICHAR[1]	MIN0	TAN
AND	CSIN	DMIN1	IDIM	MIN1	TAND
ANINT	CSQRT	DMOD	IDINT	MOD	TANH
ASIN	DABS	DNINT	IDNINT	NEQV	UBOUND[2]
ATAN	DACOS	DPROD	IFIX	NINT	XOR
ATAN2	DASIN	DSIGN	INDEX	OR	
ATANH	DATAN	DSIN	INT	RANK	

[1] Valid only if the fixed collation weight table is in effect. Collation weight tables are described in chapter 10, NOS/VE and Utility Subprograms.

[2] Valid only if the bound is known at compile time.

An extended integer constant expression is an extended arithmetic or boolean constant expression in which each operand is an integer or boolean constant, a symbolic integer or boolean constant, an extended integer constant expression enclosed in parentheses, or a reference to one of the following intrinsic functions. Arguments to the intrinsic functions must be extended constant expressions.

ABS[1]	IABS	INDEX	MAX0	NEQV
AND	ICHAR[2]	INT	MAX1	NINT
BOOL	IDIM	ISIGN	MIN[1]	OR
COMPL	IDINT	LEN	MIN0	SHIFT
DIM[1]	IDNINT	MASK	MIN1	SIGN[1]
EQV	IFIX	MAX[1]	MOD[1]	XOR

[1] Valid only with integer or boolean arguments.

[2] Valid only if the fixed collation weight table is in effect. Collation weight tables are described in chapter 10, NOS/VE and Utility Subprograms.

Type of an Arithmetic Expression

The type of an arithmetic expression containing one or more arithmetic operators is determined from the types of the operands. Integer expressions, real expressions, double precision expressions, and complex expressions are arithmetic expressions whose values are of type integer, real, double precision, and complex, respectively. When the operator + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand unless the operand is of type boolean, in which case the type of the resulting expression is integer.

When an arithmetic operator operates on a pair of arithmetic operands of the same type, the result has the same type as the operands. The following table shows the type of the result if the operands are of different types:

Table 5-1. Resulting Type for X1 ** X2

Type of x1	Type of x2	x1 Value Used	Converted x2 Value Used	Resulting Data Type
Integer	Integer	x1	x2	Integer
Integer	Real	REAL(x1)	x2	Real
Integer	Double	DBLE(x1)	x2	Double
Integer	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Real	Integer	x1	x2	Real
Real	Real	x1	x2	Real
Real	Double	DBLE(x1)	x2	Double
Real	Complex	CMPLX(x1,0.)	x2	Complex
Double	Integer	x1	x2	Double
Double	Real	x1	DBLE(x2)	Double
Double	Double	x1	x2	Double
Double	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Complex	Integer	x1	x2	Complex
Complex	Real	x1	CMPLX(x2,0.)	Complex
Complex	Double	x1	CMPLX(REAL(x2),0.)	Complex
Complex	Complex	x1	x2	Complex

Byte operands are treated the same as integer operands.

Table 5-2. Resulting Type for X1 + X2, X1 - X2, X1 * X2, or X1 / X2

Type of x1	Type of x2	x1 Value Used	Converted x2 Value Used	Resulting Data Type
Integer	Integer	x1	x2	Integer
Integer	Real	REAL(x1)	x2	Real
Integer	Double	DBLE(x1)	x2	Double
Integer	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Real	Integer	x1	REAL(x2)	Real
Real	Real	x1	x2	Real
Real	Double	DBLE(x1)	x2	Double
Real	Complex	CMPLX(x1,0.)	x2	Complex
Double	Integer	x1	DBLE(x2)	Double
Double	Real	x1	DBLE(x2)	Double
Double	Double	x1	x2	Double
Double	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Complex	Integer	x1	CMPLX(REAL(x2),0.)	Complex
Complex	Real	x1	CMPLX(x2,0.)	Complex
Complex	Double	x1	CMPLX(REAL(x2),0.)	Complex
Complex	Complex	x1	x2	Complex

Byte operands are treated the same as integer operands.

If two arithmetic operands are of different types, the operand that differs in type from the result of the operation is converted to the type of the result. The operator then operates on a pair of operands of the same type. The exception to this is an operand of type real, double precision, or complex raised to an integer power; the integer operand is not converted. If the value of J is negative, the interpretation of I ** J is the same as the interpretation of 1/(I ** ABS(J)), which is subject to the rules for integer division. For example, 2 ** (-3) has the value of 1/(2 ** 3), which is zero.

If a subexpression consists of one operator and a single operand or a pair of operands, the type of the subexpression is independent of any other operand in the expression in which it appears. For example, if X is type real, J is type integer, and INT is the real-to-integer conversion function, the expression INT(X + J) is an integer expression but X + J is a real expression.

A boolean operand in an exponentiation operation is always converted to integer. For the + - * and / operations, if two operands are of different type and one type is boolean, the result has the type of the other operand. If both operands are of type boolean, the result has type integer. The result of the operator + or the operator - operating on a single boolean operand is of type integer. A boolean operand is converted to the type of the result, and the operation is performed on the converted operand. (If you combine boolean operands with operands that are neither boolean nor integer, it can lead to unexpected results.)

One operand of type integer can be divided by another operand of type integer to yield an integer result. The result is the signed nonfractional part of the mathematical quotient. For example, (-10)/4 yields -2; the result is formed by discarding the fractional part of the mathematical quotient -2.5.

NOTE

To ensure the correct evaluation of a real, double precision, complex, or boolean expression, the operand must either be a standard normalized floating-point number or zero (represented as all zero bits). FORTRAN automatically normalizes all real non-zero constants, and the results of all floating point operations with standard normalized or zero operands are normalized or zero. However, it is possible to generate unnormalized or nonstandard operands by means of boolean expressions, equivalencing, or various input operations.

Character Expressions

The value of a character expression is a character string. A character expression consists of one or more of the following primaries separated by the character operator and parentheses:

A constant
A symbolic constant
A variable
A substring reference
An array reference
An array section reference
An array element
A function reference

The character operator is:

//

The result of the concatenation operation is a character string joined with another string, whose length is the sum of the lengths of the strings.

Evaluation of a character expression produces a result of type character. The simplest form of a character expression is one of the primaries listed above, for example, a constant of type character. You can form more complicated character expressions by using two or more character primaries together with the character operator, for example:

'AB' // 'CDE'

The value of 'AB' // 'CDE' is the string 'ABCDE' and the value of 'AB ' // 'DEF' is the string 'AB DEF'.

The operands of a character expression must identify values of type character. Except in a character assignment statement, a character expression cannot concatenate an operand whose length specification is an asterisk in parentheses, unless the operand is a symbolic constant.

Operator Precedence

In a character expression containing two or more operands, the operands are combined from left to right to evaluate the expression. For example, the interpretation of the character expression

'AB' // 'CD' // 'EF'

is the same as the interpretation of the character expression

('AB' // 'CD') // 'EF'

The value of the preceding expression is the same as that of the constant 'ABCDEF'. Note that parentheses have no effect on the value of a character expression. Thus, the expression

'AB' // ('CD' // 'EF')

has the same value as the preceding expressions.

Character Constant Expression

A character constant expression is a character expression with no variable, array, array section, array element, substring, or function references. It contains only character constants, symbolic constants, or either of these enclosed in parentheses.

Extended Character Constant Expression

An extended character constant expression is a character constant expression that also contains a reference to the CHAR intrinsic function. The argument to CHAR must be an integer constant expression. The fixed collation table must be in effect.

Logical Expressions

The value of a logical expression is one of the logical values .TRUE. or .FALSE. Logical expressions consist of one or more of the following primaries of type logical separated by operators and parentheses:

A constant
A symbolic constant
A variable
An array element reference
An array reference
An array section reference
A function reference
A relational expression (described later in this chapter)

The logical operators are:

.NOT.	Logical negation
.AND.	Logical conjunction
.OR.	Logical inclusive disjunction
.EQV.	Logical equivalence
.NEQV.	Logical nonequivalence
.XOR.	Logical exclusive disjunction

The simplest form of a logical expression is one of the primaries listed above; for example, a constant or variable of type logical. You can form a more complicated logical expression by using one or more logical operands together with logical operators and parentheses, for example:

```
.NOT. L

.NOT. (X .GT. Y)

x .gt. y .and. .not. z
```

Examples of invalid logical expressions:

`.AND. M`	.AND. must be both preceded and followed by a logical expression.
`L .AND. .OR. M`	.AND. must be separated from .OR. by a logical expression.

The following table shows the value of a logical operation involving each logical operator:

Value x1 Value x2	.TRUE. .TRUE.	.TRUE. .FALSE.	.FALSE. .TRUE.	.FALSE. .FALSE.
.NOT.x2	.FALSE.	.TRUE.	.FALSE.	.TRUE.
x1.AND.x2	.TRUE.	.FALSE.	.FALSE.	.FALSE.
x1.OR.x2	.TRUE.	.TRUE.	.TRUE.	.FALSE.
x1.EQV.x2	.TRUE.	.FALSE.	.FALSE.	.TRUE.
x1.NEQV.x2	.FALSE.	.TRUE.	.TRUE.	.FALSE.
x1.XOR.x2	.FALSE.	.TRUE.	.TRUE.	.FALSE.

Control Data Extension

Variable-Length Logical Expressions

The length of a logical constant expression containing items of different lengths is the length of the longest item. For example, given the declarations

```
LOGICAL L*2, A*2, J*1, E*8
```

the following expression lengths exist:

Expression	Length
A .AND. E	8 bytes
J .OR. L	2 bytes
.NOT. J	1 byte

End of Control Data Extension

Operator Precedence

Operator precedence establishes how a logical expression with two or more logical operators is interpreted. The following operator precedence determines the order in which the operands are combined:

.NOT.	Highest (evaluated first)
.AND.	
.OR.	
.EQV. or .NEQV. or .XOR.	Lowest (evaluated last)

For example, in the expression:

```
A .OR. B .AND. C
```

the .AND. operator has higher precedence than the .OR. operator; therefore, the expression is evaluated the same as:

```
A .OR. (B .AND. C)
```

When a logical expression contains two or more .AND. operators; two or more .OR. operators; or two or more .EQV., NEQV., or .XOR. operators, the logical quantities are combined from left to right.

You can alter the default order of evaluation of subexpressions within an expression by enclosing the subexpression in parentheses. The subexpression within the parentheses is always evaluated before being combined with other operands in the expression. For example, in the expression

```
(A .OR. B) .AND. C
```

the result of A .OR. B is used to perform the logical conjunction with C.

Logical Constant Expression

A logical constant expression is a logical expression with no variable, array, array section, array element, or function references. It contains only logical constants or symbolic constants or either of these enclosed in parentheses.

Extended Logical Constant Expression

An extended logical constant expression is a logical constant expression that also contains a reference to any of the following intrinsic functions. Arguments to the intrinsic functions must be extended integer constant expressions.

LGE
LGT
LLE
LLT

Control Data Extension

Boolean Expressions

A boolean expression is an expression whose value is boolean. Boolean expressions consist of one or more of the following primaries of boolean or arithmetic type separated by logical operators and parentheses:

An unsigned constant
A symbolic constant
A variable
An array reference
An array section reference
An array element reference
A function reference
An arithmetic expression

The boolean operators are the logical operators:

.NOT.	Logical negation
.AND.	Logical conjunction
.OR.	Logical inclusive disjunction
.EQV.	Logical equivalence
.NEQV.	Logical nonequivalence
.XOR.	Logical exclusive disjunction

The simplest boolean expression is one of the primaries listed above; for example, a boolean constant or an arithmetic expression. You can form more complicated boolean expressions using one or more of the primaries listed above together with the boolean operators and parentheses, for example:

(P, Q, R, S, X, Y and Z can be any type except character or logical.)

```
X .AND. Z"FFFF"

(X + Y) .OR. R

P .AND. Q .OR. S
```

Control Data Extension *(Continued)*

A boolean operator determines each bit value of the result it yields independently of other bits of the result. Each bit value of the result is determined from the corresponding bit values of the operands. At each bit position, the bit in the result is determined as shown in the following table:

Each Bit in x1	Corresponding Bit in x2	Corresponding Bit in Result of					
		.NOT.x2	x1.AND.x2	x1.OR.x2	x1.EQV.x2	x1.NEQV.x2	x1.XOR.x2
0	0	1	0	0	1	0	0
0	1	0	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	0	1	1	1	0	0

Boolean expressions containing 1-, 2-, 4-byte integers are considered 8-bytes long.

Operator Precedence

Boolean quantities are combined from left to right when a boolean expression contains two or more .AND. operators, two or more .OR. operators, or two or more .EQV., .NEQV., or .XOR. operators.

Boolean Constant Expression

A boolean constant expression is a boolean expression with no variable, array, array section, array element, or function references. It contains only boolean constants, symbolic boolean constants, arithmetic constants, symbolic arithmetic constants, or any of the preceding enclosed in parentheses.

Extended Boolean Constant Expression

An extended boolean constant expression is a boolean constant expression that also contains a reference to any of the intrinsic functions listed for arithmetic constant expressions with constant expressions as actual arguments.

Data Type of a Boolean Expression

If an operand is of arithmetic type (integer, real, double precision, or complex) it is converted to boolean and the operation is performed on the converted operand. The conversion of an integer or real value to boolean does not change its bit pattern.

End of Control Data Extension

Relational Expressions

Relational expressions can appear only within logical expressions. The value of a relational expression is one of the logical values .TRUE. or .FALSE. Relational expressions consist of one or more of the following primaries separated by operators:

An arithmetic expression
A boolean expression
A character expression

The relational operators are:

.LT.	less than (<)
.LE.	less than or equal to (≤)
.EQ.	equal to (=)
.NE.	not equal to (P)
.GE.	greater than or equal to (≥)
.GT.	greater than (>)

You cannot use a relational expression to compare the value of an arithmetic expression with the value of a character expression. For two operands A and B, the relational operators have the following meaning:

A .LT. B	Is A less than B?
A .LE. B	Is A less than or equal to B?
A .EQ. B	Is A equal to B?
A .NE. B	Is A not equal to B?
A .GT. B	Is A greater than B?
A .GE. B	Is A greater than or equal to B?

You can use a complex operand only when the relational operator is .EQ. or .NE. because there is no mathematically accepted ordering for complex values.

The result of a relational expression is an 8-byte logical value, regardless of the length of the primaries in the expression.

NOTE

You should ensure that real operands in relational expressions involving the .EQ. or .NE. operators contain normalized standard floating-point numbers or zero (represented as all zero bits) if a floating-point comparison is desired. A bit-by-bit comparison is performed for these operators in order to allow comparisons of Hollerith data in real variables. This means that two different unnormalized representations of the same floating-point value compares as not equal.

FORTRAN automatically normalizes all real nonzero constants, and the results of all floating-point operations with standard normalized or zero operands are normalized or zero. However, it is possible to generate unnormalized or nonstandard operands by means of boolean expressions, equivalencing, or various input operations.

Arithmetic and Boolean Relational Expressions

An arithmetic or boolean relational expression has the logical value .TRUE. only if the values of the operands satisfy the relation specified by the operator. If the two arithmetic expressions are of different types, or both are boolean, the value of the relational expression

X1 relop X2

is the value of the expression

((X1) - (X2)) relop 0

where 0 (zero) is of the same type as the expression.

Examples of valid relational expressions (assume that J and ITEM are type integer, and VAR, B, and C are type real):

J .LT. ITEM	Is J less than ITEM?
580.2 .gt. var	Is 580.2 greater than VAR?
B .EQ. (2.7, 59E3)	Only the real part of complex operand is used.
C .LT. 1.5D4	Is DBLE(C) less than 1.5D4?

Example of invalid relational expression:

A .GT. 720 .LT. 900	Two relational operands are not permitted in one expression.

Character Relational Expressions

A character relational expression has the logical value .true. only if the values of the operands satisfy the relation specified by the operator. The character expression X1 is considered to be less than X2 if the value of X1 precedes the value of X2 in the collating sequence; X1 is greater than X2 if the value of X1 follows the value of X2 in the collating sequence. The collating sequence in use determines the result of the comparison. The default collating sequence is the ASCII collating sequence as shown in appendix C. To select another collating sequence, see the Collation Control Subprograms in chapter 10, NOS/VE and Utility Subprograms.

Character relational expressions in PARAMETER and conditional compilation (C$ IF) statements are always evaluated using the ASCII sequence.

If the operands are of unequal length, the shorter operand is treated as if it had been extended on the right with spaces to the length of the longer operand.

General Rules for Expressions

The order in which operands are evaluated using operators is determined by:

1. Use of parentheses
2. Precedence of the operators
3. Right-to-left interpretation of exponentiations
4. Left-to-right interpretation of multiplications and divisions
5. Left-to-right interpretation of additions and subtractions in an arithmetic expression
6. Left-to-right interpretation of concatenations in a character expression
7. Left-to-right interpretation of .AND. operators
8. Left-to-right interpretation of .OR. and .NOT. operators
9. Left-to-right interpretation of .EQV., NEQV., and .XOR. operators in a logical expression or boolean expression

Precedences have been established among the arithmetic and logical operators. There is only one character operator. No precedence is established within the relational operators. The precedences among the operators are:

Arithmetic	Highest
Character	
Relational	
Logical	Lowest

An expression can contain more than one kind of operator. For example, the logical expression

```
L .OR. A + B .GE. C
```

where A, B, and C are of type real, and L is of type logical, contains an arithmetic operator(+), a relational operator(.GE.), and a logical operator(.OR.). This Expression would be interpreted as:

```
L .OR. ((A + B) .GE. C)
```

Any variable, array, array section, array element, function, or character substring referenced in an expression must be defined at the time the reference is made. An integer operand must be defined with an integer value rather than an assigned statement label value. If you reference a character string or substring, all of the character positions you reference must be defined at the time the reference is executed.

You must not specify any arithmetic operation whose result is not mathematically defined; for example, dividing by zero, or raising a zero value to a zero-valued or negative-valued power.

A function reference in a statement must not alter the value of any other entity within the same statement. The execution of a function reference in a statement must not alter the value of any entity in common that affects the value of any other function reference in that statement. However, execution of a function reference in the expression of a logical IF statement can affect entities in the statement that is executed when the value of the expression is true. If a function reference causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example:

`A(I) = F(I)` `Y = G(X) + X`	These statements are prohibited if the reference to F defines I, or the reference to G defines X.

The data type of an expression which contains a function reference does not affect the evaluation of the actual arguments of the function. However, the result of a generic function reference assumes a data type that depends on the data type of its arguments.

If an array element reference is executed, the subscript is evaluated. The data type of an expression which contains a subscript does not affect, nor is it affected by, the evaluation of the subscript.

If a substring reference is executed, its substring expression is evaluated. The data type of an expression which contains a substring name does not affect, nor is it affected by, the evaluation of the substring expression.

All of the operands of an expression are not necessarily evaluated if the value of the expression can be determined otherwise. For example, in the logical expression

X .GT. Y .OR. L(Z)

where X, Y, and Z are real, and L is a logical function, the function reference L(Z) need not be evaluated if X is greater than Y.

If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference. In the example

X .GT. Y .OR. L(Z)

the evaluation of the expression causes Z to become undefined if L defines its argument.

If a statement contains more than one function reference, the functions can be evaluated in any order, except for a logical IF statement and a function argument list containing function references. For example, the statement

Y = F(G(X))

where F and G are functions, requires G to be evaluated before F is evaluated.

Any expression contained in parentheses is always treated as an entity. For example, in the expression A * (B * C), the product of B and C is evaluated and then multiplied by A; the mathematically equivalent expression (A * B) * C is not used.

A non-integer arithmetic expression contained in parentheses is always treated as an entity. An integer arithmetic expression is treated as an entity if the EXPRESSION_EVALUATION parameter specifies CANONICAL or MAINTAIN_EXCEPTIONS on the VECTOR_FORTRAN command.

Control Data Extension

Array-Valued Expressions

An array-valued expression includes an array, array section, or an intrinsic function with an array result.

Array-valued expressions must not appear:

- As a logical expression of a logical IF statement.
- As a logical expression of a block IF or ELSE IF statement.
- As initial, terminal, or increment expressions of a DO statement.
- As an arithmetic expression in an arithmetic IF statement.
- In a computed GOTO, OPEN, CLOSE, INQUIRE, or RETURN statement.
- In the REC= specifier on a READ, WRITE, PRINT, or PUNCH statement.
- As an external unit identifier.
- As constants.

If an array section reference is executed, its section subscript expression is evaluated. The data type of an expression that contains an array section does not affect, nor is affected by, the evaluation of the array section. When an arithmetic, character, relational, or logical operator operates on two operands where at least one of the operands is an array, the operands must be conformable. The result is the same shape as the array operand or array operands.

For example, the array expression

```
A + B
```

produces an array the same shape as A and B. The individual array elements of the result have the values of the first element of A added to the first element of B, the second element of A added to the second element of B, and so forth.

End of Control Data Extension

Assignment Statements

Assignment statements are executable statements that use an expression to define or redefine the values of variables or arrays.

An assignment statement consists of a variable, array, array section, array element, or substring followed by an equal sign followed by an expression. (In the case of multiple assignment statements, multiple equal signs and multiple variables are allowed.) When the assignment statement is executed, the expression is evaluated, and the result is stored in the variable, array, array section, array element, or substring. If the variable has been previously defined, the new value replaces the current value. There are six types of assignment statements:

Arithmetic

Character

Logical

Boolean

Multiple

Statement label (with the ASSIGN statement as described in chapter 6, Flow Control Statements).

Arithmetic Assignment Statement

An arithmetic assignment statement has the form:

v = exp

v

A variable, array, array section, or array element of type integer, byte, real, double precision, or complex

exp

An arithmetic or boolean expression

If **exp** is an array, array section name, or array-valued expression, it must be conformable with **v**. If **v** is a scalar, **exp** must be a scalar. If **v** is an array or array section and **exp** is a scalar, **exp** is extended to an array matching the dimensions of **v** in which all elements have the value of **exp**.

After the arithmetic or boolean expression **exp** is evaluated, the result is converted to the type of **v** in the following way:

Integer, Byte	INT(exp)
Real	REAL(exp)
Double precision	DBLE(exp)
Complex	CMPLX(exp)

The result is then assigned to **v**, and **v** is defined or redefined with that value. If the type of **v** is integer, and the length of **v** is less than the length of **exp**, then **exp** is truncated on the left to match the length of **v**. The truncation can change the sign of **exp**.

Examples:

`A = A + 1.0`	A is incremented by 1.
`VAR = VALUE + (7/4) * 32`	VAR is assigned the value of VALUE + (7/4 * 32)
`WAGE(2:5) = PAY(1:4) - TAX(4)`	The array section WAGE(2:5) is assigned the value of the array section PAY(1:4) - TAX(4). Since TAX(4) is a scalar value, it is treated as an array of size 4 with all elements having the value TAX(4).

```
COMPLEX C
PARAMETER(PAR1 = 1., PAR2 = 2.)
      :
C = (PAR1, PAR2)
```

A value is assigned to the complex variable C.

```
DIMENSION ERROR_POINTS(2, 5)
ERROR_POINTS(:, 4:5) = 0.0
      :
```

An array section of array ERROR_POINTS is initialized to zeros.

Examples of invalid arithmetic assignment statements:

```
B + C = X - 5.1
```

An expression must not appear to the left of the equal sign.

```
DIMENSION A(2,2), B(4)
  :
A=B
```

The array objects in the assignment statement must be conformable.

Character Assignment Statement

A character assignment statement has the form:

v = exp

v

A character variable, array, array section, array element, or substring.

exp

A character expression.

The character expression **exp** is evaluated, and the result is then assigned to **v**. If the character positions being defined in **v** are referenced in **exp**, you must specify EXPRESSION_EVALUATION = OVERLAPPING_STRING_MOVES on the VECTOR_FORTRAN command.

If **exp** is an array, array section name, or array-valued expression, it must be conformable with **v**. If **v** is a scalar, **exp** must be a scalar. If **v** is an array or array section and **exp** is a scalar, **exp** is extended to an array matching the dimensions of **v** in which all elements have the value of **exp**.

The variable **v** and expression **exp** can have different lengths. If the length of **v** is greater than the length of **exp**, the effect is as though **exp** were extended to the right with spaces until it is the same length as **v**. If the length of **v** is less than the length of **exp**, the effect is as though **exp** were truncated from the right until it is the same length as **v**.

If **v** is a substring, **exp** is assigned only to the substring. Substrings not specified by **v** are unchanged.

Example:

`CHARACTER MO*3,DAY*3,DATE*10`
`⋮`
`DATE = MO//DAY//'1988'`
	The character variables MO and DAY and the string 1988 are concatenated into a single 10-character string and stored into the variable DATE.

Only as much of the value of **exp** must be defined as is needed to define **v**. For example:

`CHARACTER A*2, B*4` `A = B`	The assignment A = B requires that the substring B(1:2) be defined. It does not require that the substring B(3:4) be defined.

Logical Assignment Statement

A logical assignment statement has the form:

v = exp

v

A logical variable, array, array section, or array element of any length.

exp

A logical expression of any length.

The logical expression is evaluated, and the result is then assigned to **v**. **Exp** must have a value of either true or false.

If **exp** is an array, array section name, or array-valued expression, it must be conformable with **v**. If **v** is a scalar, **exp** must be a scalar. If **v** is an array or array section and **exp** is a scalar, **exp** is extended to an array matching the dimensions of **v** in which all elements have the value of **exp**.

Example:

```
LOGICAL LOG2
I = 1
LOG2 = I .EQ. 0
```

LOG2 is assigned the value .FALSE. because I is not equal to zero.

Control Data Extension

Boolean Assignment Statement

A boolean assignment statement has the form:

v = exp

v

A boolean variable, array, array section, or array element.

exp

A boolean or arithmetic expression.

The boolean or arithmetic expression **exp** is evaluated. If **exp** is an arithmetic expression, the result used is BOOL(exp). The result is then assigned to **v**.

If **exp** is an array, array section name, or array-valued expression, it must be conformable with **v**. If **v** is a scalar, **exp** must be a scalar. If **v** is an array or array section and **exp** is a scalar, **exp** is extended to an array matching the dimensions of **v** in which all elements have the value of **exp**.

Example:

`A = B .AND. Z"FF"`	The lowest (rightmost) 8 bits of B are stored in A; the upper 56 bits of A contain zeros.

End of Control Data Extension

Control Data Extension

Multiple Assignment Statement

A multiple assignment statement has the form:

v = ... = *v* = **exp**

v

A variable, array, array section, array element, or character substring.

exp

An arithmetic, boolean, character, relational, or logical expression. The type of **exp** must ensure that **v** = **exp** is valid for each **v** specified.

If **exp** is an array, array section name, or array-valued expression, it must be conformable with **v**. If **v** is a scalar, **exp** must be a scalar. If **v** is an array or array section and **exp** is a scalar, **exp** is extended to an array matching the dimensions of **v** in which all elements have the value of **exp**.

Execution of a multiple assignment statement causes the evaluation of the expression **exp**. After any necessary conversion, the assignment and definition of the rightmost **v** with the value of **exp** occurs. Assignment and definition of each additional **v** occurs in right-to-left order. The value assigned to each **v** is the value of the **v** immediately to its right, after any necessary conversion.

If the type of **v** is integer, and the length of **v** is less than the length of **exp**, then **exp** is truncated on the left to match the length of **v**. The truncation can change the sign of **exp**.

Example:

`X = M = 1.5`	M is assigned the value 1 and X is assigned the value 1.0.

Example:

`REAL APPLES(4), ORANGES(10)`
`:`
`APPLES = ORANGES(2:5) = 7.0`
	Elements 2, 3, 4, and 5 of array ORANGES are assigned the value 7.0 and the entire array APPLES is assigned the value 7.0. Elements 1, 6, 7, 8, 9 and 10 or array ORANGES are unchanged.

End of Control Data Extension

Flow Control Statements 6

GO TO Statements 6-1
Unconditional GO TO Statement 6-1
Computed GO TO Statement 6-2
Assigned GO TO Statement 6-3
ASSIGN Statement 6-4

IF Statements 6-5
Arithmetic IF Statement 6-5
Logical IF Statement 6-6
Block IF Structures 6-7
Nested Block IF Structures 6-9

WHERE Statements 6-10
Logical WHERE Statement 6-10
Block WHERE Structures 6-11

DO Statement 6-13
DO Loops 6-13
Nested DO Loops 6-16
Extended Range DO Loops 6-20

CONTINUE Statement 6-21

PAUSE Statement 6-22

STOP Statement 6-23

END Statement 6-24

Flow Control Statements 6

The flow control statements change the order in which statements execute. The flow control statements described in this chapter are:

GO TO
IF
WHERE
DO
CONTINUE
PAUSE
STOP
END

The CALL and RETURN statements, sometimes considered to be flow control statements, are described in chapter 8, Program Units.

GO TO Statements

The three types of GO TO statements are:

Unconditional GO TO

Computed GO TO

Assigned GO TO

The ASSIGN statement is used in conjunction with the assigned GO TO and is also described with the assigned GO TO statement.

Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the executable statement identified by the specified label. The unconditional GO TO statement has the form:

GO TO label

label
Label of an executable statement

The labeled statement must appear in the same program unit as the GO TO statement.

Example:

```
   GO TO 20
10 A = 0.0
20 A = A + 1.0
     :
```

When the statement GO TO 20 is reached, control transfers to statement 20. The statement following the GO TO can be reached only as a result of another flow control statement or the ERR= or END= specifiers on the input/output status statements.

Computed GO TO Statement

The computed GO TO statement transfers control to the executable statement identified by one of the specified labels. The computed GO TO statement has the form:

GO TO (label, ..., *label*), **exp**

label

Label of an executable statement that appears in the same program unit as the GO TO statement.

exp

An arithmetic or boolean expression. (ANSI allows an integer expression only.) The comma preceding **exp** is optional.

The value of the expression determines which label is selected. If **exp** has a value of one, control transfers to the statement identified by the first label in the list; if **exp** has a value of *i*, control transfers to the statement identified by the *ith* label in the list. The value of **exp** is truncated and converted to integer, if necessary.

If the value of **exp** is less than one or greater than the number of labels in the list, execution continues with the statement following the computed GO TO.

Example:

```
GO TO (10, 20, 30, 20) L
```

The next statement executed is:

10 if L=1
20 if L=2
30 if L=3
20 if L=4

Example:

```
M=4
GO TO(100, 200, 300) M
A=B + C
```

Execution continues with the statement A = B + C because the value of M is greater than the number of labels in parentheses.

Assigned GO TO Statement

An assigned GO TO uses an assigned GO TO statement and an ASSIGN statement. The ASSIGN statement is used to assign a label to a variable. The variable is used in the assigned GO TO statement.

The assigned GO TO statement transfers control to the executable statement whose label was last assigned to **iv** by the execution of a prior ASSIGN statement. The assigned GO TO statement has the form:

GO TO iv, *(label, ..., label)*

iv

8-byte integer variable. The comma following **iv** is optional.

label

Optional; label of an executable statement that appears in the same program unit as the assigned GO TO statement. The list of labels, including parentheses and preceding comma, can be entirely omitted.

At the time of execution of an assigned GO TO statement, the value of variable **iv** must be a statement label of an executable statement that appears in the same program unit. All labels in a statement label list must be in the same program unit as both the ASSIGN and assigned GO TO statements. The label assigned to iv need not be in the list.

Example:

```
   ASSIGN 50 TO JUMP
10 GO TO JUMP(20, 30, 40, 50)
20 CONTINUE
     :
30 CAT = ZERO + HAT
     :
   40  CAT = ZERO + RAT
     :
50 CAT = ZERO + BAT
```

Statement 50 is executed immediately after statement 10.

ASSIGN Statement

The ASSIGN statement assigns a statement label to an integer variable. The ASSIGN statement has the form:

ASSIGN label TO iv

label

Label of an executable or FORMAT statement

iv

8-byte integer variable

The value assigned to **iv** represents the label of an executable or a FORMAT statement. The labeled statement must appear in the same program unit as the ASSIGN statement. When defined by an ASSIGN statement, **iv** can be referenced only in an assigned GO TO statement or an input/output format specifier.

The assignment must be made prior to execution of the assigned GO TO statement or the input/output statement that references **label**.

Example:

```
   ASSIGN 10 TO LSWIT
   GO TO LSWIT(5, 10, 15, 20)
        :
10 BLUE_JAYS = 3.0 + BIRD4
```

At the GO TO statement, control transfers to the statement labeled 10.

Example:

```
   ASSIGN 24 TO IFMT
   WRITE(2, IFMT) A, B
        :
24 FORMAT(F3.2, F5.6)
```

The variables A and B are formatted according to the FORMAT statement labeled 24.

IF Statements

The IF statement evaluates an expression and conditionally transfers control or executes another statement, depending on the result of the expression. The types of IF statements are:

Arithmetic IF
Logical IF
Block IF

The ELSE, ELSE IF, and END IF statements are used in conjunction with a block IF statement and are described in this chapter under Block IF Structures.

Arithmetic IF Statement

The arithmetic IF statement evaluates an arithmetic expression and conditionally transfers control to another statement. The arithmetic IF statement has the form:

IF(exp) label1, label2, label3

exp

Integer, real, double precision, or boolean expression.

label1, label2, and **label3**

Label of an executable statement that appears in the same program unit as the arithmetic IF statement.

The arithmetic IF statement transfers control to the statement labeled **label1** if the value of **exp** is less than zero, to the statement labeled **label2** if it is equal to zero, or to the statement labeled **label3** if it is greater than zero. If **exp** is type boolean, INT(**exp**) is used.

Example:

```
      IF(I - N) 3, 4, 5
3     ISUM = J + K
      GO TO 4
5     CALL ERROR1
4     PRINT *, ISUM
      STOP
```

If I is less than N, control transfers to statement 3; if I is equal to N, control transfers to statement 4; if I is greater than N, control transfers to statement 5.

Logical IF Statement

The logical IF statement evaluates a logical expression and conditionally executes another statement. The logical IF statement has the form:

IF(exp) statement

exp

Logical expression

statement

Any executable statement except a DO, block IF, ELSE, ELSE IF, END, END IF, or another logical IF, block WHERE, ELSEWHERE, ENDWHERE, or logical WHERE statement.

If the value of **exp** is true, **statement** is executed. If the value of **exp** is false, **statement** is not executed; execution continues with the next statement.

Example:

```
    IF(P .AND. Q) RES = 7.2
50  TEMP = ANS * Z
```

If P and Q both have the logical value true, RES is set to 7.2; otherwise, RES is unchanged. In either case, statement 50 is executed.

Example:

```
    IF(A .LT. B) GO TO 50
20  Z = T + R
     :
50  Z = Z + 1.0
```

If A is less than B, control transfers to statement 50; otherwise, execution continues with statement 20.

Block IF Structures

Block IF structures allow you to execute different blocks of statements. A block IF structure has the form:

IF(exp) THEN

if-block

ELSE IF(exp) THEN

if-block

⋮

ELSE IF(exp) THEN

if-block

ELSE

if-block

END IF

In a block IF structure with ELSE IF statements, the initial block IF statement and each ELSE IF or ELSE statement has an associated if-block. At most one of these if-blocks is executed as follows, even if more than one of the specified conditions is satisfied:

1. Each logical expression is evaluated, in order of source statements, until one is found that has the value true.
2. Control then transfers to the first statement of the associated if-block.
3. When execution of the if-block has completed, and if control has not been transferred outside the if-block, execution continues with the next statement following the ENDIF statement.
4. If none of the logical expressions has the value true, control transfers to the statement following the ELSE statement and executes the associated if-block.
5. If there is no ELSE statement, control transfers to the statement following the END IF statement.

The if-blocks can contain any number of executable statements, including other block IF statements.

Control can transfer out of a block IF structure from inside any if-block; however, control cannot transfer from one if-block to another if they are at the same nesting level.

Control cannot be transferred into an if-block from outside the if-block. An if-block can be entered only through its associated block IF, ELSE IF, or ELSE statement. You cannot transfer control directly to an ELSE, ELSE IF, or END IF statement. However, you can transfer control directly to a block IF statement.

In examples in this manual, statements within block IF structures are indented. Although this is not a requirement, it is recommended to improve clarity in your programs.

An example of a simple block IF structure is:

```
IF(I .EQ. 0) THEN
    X = X + DX
    Y = Y + DY
END IF
PRINT 10, X, Y
```

If I is zero, the two succeeding statements are executed; otherwise, the statements are not executed. In either case, execution continues with the statement following END IF.

An example of a block IF structure with one ELSE statement is:

```
IF(A .LT. B) THEN
    X = A + B
    XSUM = XSUM + X
ELSE
    Y = A * B
    YSUM = YSUM + Y
ENDIF
PRINT 15, X, Y
```

If A is less than B, new values for X and XSUM are calculated; otherwise, new values for Y and YSUM are calculated. In either case, the values of X and Y are printed.

An example of a block IF structure with two ELSE IF statements and an ELSE statement is:

```
IF(N .EQ. 2) THEN
    X = 1.0
    Y = 2.0
ELSE IF (N .EQ. 3) THEN
    X = X + 10.0
    Y = Y + 10.0
ELSE IF(N .LT. 0) THEN
    X = 0.0
    Y = 0.0
ELSE
    CALL ERRSUB
ENDIF
```

If N equals 2, the X and Y are assigned 1.0 and 2.0. IF N equals 3, X and Y are incremented by 10.0. If N is less than 0, X and Y are assigned 0.0. If N does not satisfy any of these conditions, an error-processing subroutine named ERRSUB is called.

Nested Block IF Structures

Block IF structures can be nested; that is, any if-block within a structure can itself contain block IF structures. Within a nesting hierarchy, control can transfer from an inner level structure into an outer level structure; however, control cannot transfer from an outer level structure into an inner level structure.

The general form of a nested block-IF structure with two levels of nesting is:

IF(exp) THEN

if-block

IF(exp) THEN

if-block

END IF

END IF

Nested block IF statements cannot share END IF statements; each block IF statement requires its own END IF statement.

Example:

```
IF(X .GT. Y) THEN
    Y = Y + YINCR
    IF(K .EQ. J) THEN
        XT = X
        YT = Y
    ELSE
        K = K + 1
    END IF
ELSE
    X = X + XINCR
END IF
```

The preceding structure contains two levels of nesting. Each level contains a block IF and an ELSE statement. The inner structure is executed only if X is greater than Y.

Control Data Extension

WHERE Statements

WHERE statements control the assignment of values in array assignment statements according to a logical expression. The two types of WHERE statements are logical WHERE and block WHERE.

Logical WHERE Statement

The logical WHERE statement controls the assignment of values in an array assignment statement and the evaluation of expressions in the array assignment statement according to the value of a logical array expression. The logical WHERE statement has the form:

WHERE(exp) statement

exp

A logical array expression.

statement

An array assignment statement of the form

v = ... *v* = **e**

v

An array or array section reference with the same shape as **exp**.

exp

An expression of the same type as **v**.

For every element in **exp** that is evaluated to true, the corresponding element in **e** of the assignment statement is evaluated and assigned to the corresponding element of **v**. For every element in **exp** that is evaluated to false, the corresponding element of **v** is not changed and **e** is not evaluated.

If **e** is a scalar, the scalar value is treated as if it had been extended to an array with the shape of **v**, in which all elements have the value **e** before assignment occurs.

The array assignment statement must not reference array-valued intrinsic functions or the RANF or MERGE intrinsic functions.

Example:

```
INTEGER NSUM(5)
LOGICAL LOGA(5)
DATA LOGA, NSUM /5*.TRUE., 5*0/
LOGA(4) = .FALSE.
WHERE(LOGA) NSUM = 5
```

The WHERE statement assigns the value 5 to NSUM(1), NSUM(2), NSUM(3), and NSUM(5) since those elements correspond to true elements of array LOGA. The value of NSUM(4) is unchanged.

Control Data Extension *(Continued)*

Block WHERE Structures

Block WHERE structures consist of a block WHERE statement, a where-block, an ENDWHERE statement, and, optionally, an ELSEWHERE statement and ELSEWHERE block.

A where-block is all executable statements between the block WHERE statement and the next ELSEWHERE, or if no ELSEWHERE exists, the ENDWHERE statement. An elsewhere-block is all executable statements between the ELSEWHERE statement and the ENDWHERE statement.

The form of a block WHERE structure is as follows:

WHERE (exp)

where-block

ELSEWHERE

elsewhere-block

ENDWHERE

All of the executable statements in a where-block or an elsewhere-block must be array assignment statements of the form:

v = ... *v* = **e**

v

An array or array section reference with the same shape as **exp**.

e

An expression of the same type as **v**.

When the WHERE statement is executed, **exp** is evaluated. For every element in **exp** that is evaluated to true, the corresponding element in **e** of the first assignment statement in the where-block is assigned to the corresponding element of **v** of the first assignment statement. For every element in **exp** that is evaluated to false, the corresponding element of **v** of the first assignment statement is not altered and **e** is not evaluated.

This process continues with each array assignment statement in sequential order. Execution continues with the statement following the ELSEWHERE statement. If the ELSEWHERE statement is not present, execution continues with the statement following the ENDWHERE statement.

For every element in **exp** that is evaluated to false, the corresponding element in **e** of the first assignment statement in the elsewhere-block is assigned to the corresponding element of **v**. For every element in **exp** that is evaluated to true, the corresponding element of **v** of the first assignment statement in the elsewhere-block is not altered and **e** is not evaluated.

The process continues with every array assignment in the elsewhere-block in sequential order until an ENDWHERE statement is reached.

Control Data Extension *(Continued)*

If **e** is a scalar in the where- or elsewhere-block, the scalar value is treated as if it had been extended to an array with the shape of **v** in which all elements have the value **e** before assignment occurs.

The array assignment statements must not reference array-processing intrinsic functions or the RANF or MERGE intrinsic functions.

A where-block or an elsewhere-block cannot contain the final statement of a DO loop or another block WHERE statement. The value of **exp** is not affected by the execution of statements in the where-block.

Control cannot be transferred into a where-block from outside the where-block. A where-block can be empty. Values defined in **v** in one assignment statement can be referenced or defined in subsequent assignment statements.

Example:

```
LOGICAL LRAY(10)
DIMENSION J(10), K(10)
DATA LRAY /10 * .FALSE./
DATA J, K /10 * 0, 10 * 0/
LRAY(1:10:2) = .TRUE.
WHERE(LRAY)
    J = J + 1
    K = K + 2
ELSEWHERE
    J = J - 1
    K = K - 2
ENDWHERE
```

Every other element of array J, beginning with J(1), is incremented by one because those values correspond to true elements of LRAY. The remaining elements of J are decremented by one because they correspond to false elements of LRAY. Similarly, every other element of array K is incremented by two and the remaining elements are decremented by two.

Example:

```
DIMENSION EMP_NUM(100)
DIMENSION RANK(100)
CHARACTER*2 CODE(100)
  :
WHERE(EMP_NUM .LT. 5000.)
    CODE = 'PT'
    RANK = RANK / 2.
ELSEWHERE
    CODE = 'FT'
    RANK = RANK + 1.
ENDWHERE
```

Elements of the character array CODE are assigned the value 'PT' if the corresponding element of EMP_NUM is less than 5000. Elements of the character array CODE contains the value 'FT' if the corresponding element of EMP_NUM is greater than or equal to 5000.

End of Control Data Extension

DO Statement

The DO statement is used to repeat execution of a group of statements. The DO statement has the form:

DO label, var = exp1, exp2, *exp3*

label

Label of an executable statement that is the final statement of the DO loop. The comma following label is optional.

var

Integer, real, or double precision variable called the DO variable.

exp1

Initial parameter.

exp2

Terminal parameter.

exp3

Increment parameter that must not equal zero. If *exp3* is omitted, the increment is 1 (the preceding comma must also be omitted).

Exp1, exp2, and exp3 are called indexing parameters; they can be integer, real, double precision, or boolean expressions.

DO Loops

The DO statement and the group of statements to be repeated are referred to as a DO loop. The DO statement determines the range of the DO loop (that is, the statements to be included in the loop) and the number of times the DO loop is to be repeated.

The final statement of a DO loop is an executable statement that must physically follow and reside in the same program unit as its associated DO statement. The final statement must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, logical WHERE, ELSEWHERE, ENDWHERE, block WHERE, STOP, END, or DO statement.

If the final statement is a logical IF statement, it must not contain a block IF, ELSE IF, END IF, ELSE, END, DO, or another logical IF statement.

NOTE

Do not alter the value of the DO variable within the body of the DO loop. Altering the value of the DO variable could cause the DO loop to repeat an incorrect number of times.

The range of a DO loop consists of all the executable statements following the DO statement up to and including the final statement. Execution of a DO statement causes the following sequence of operations to occur:

1. The expressions **exp1, exp2,** and *exp3* are evaluated and, if necessary, converted to the type of the DO variable **var**.

2. DO variable **var** is assigned the value of **exp1**.

3. The iteration count is established; this value is determined by the following expression:

   MAX(INT((m2 - m1 + m3) / m3), mtc)

   m1, m2, m3

   Values of **exp1, exp2**, and *exp3* respectively, after conversion to the type of **var**. The incrementation parameter *m3* must not equal zero.

   mtc

   Minimum trip count; *mtc* has a value of either one or zero, and is established by the ONE_TRIP_DO parameter on the VECTOR_FORTRAN command or the C$ DO directive. A zero trip count in ONE_TRIP_DO executes the loop one time. In ANSI FORTRAN, *mtc* has a value of zero.

4. If the iteration count is not zero, the range of the DO loop is executed. If the iteration count is zero, execution continues with the statement following the final statement of the DO loop; **var** retains its most recent value.

5. DO variable **var** is incremented by the value of *exp3*.

6. The iteration count is decremented by one.

Steps 4 through 6 are repeated until the iteration count in step 3 equals zero.

The iteration count of a DO loop must not exceed the range for the type of the DO variable. For example, if the DO variable is an 8-byte integer, the following conditions must be satisfied:

$| m1 + m3 | < (2 ** 63) - 1$

$| m2 + m3 | < (2 ** 63) - 1$

$| m2 - m1 | < (2 ** 63) - 1$

NOTE

When type real or double precision indexing parameters are used, accumulated roundoff error (truncation from the INT function) in the indexing calculations can cause a loop to execute more times than expected.

If a DO loop appears within an if-block, the DO loop must be entirely contained within that if-block. If a block IF statement appears within the range of a DO loop, the corresponding END IF statement must also appear within the range of that DO loop.

If a DO loop executes zero times, the DO variable value is equal to *m1* following the loop. Otherwise, the value is the most recent value of the DO variable plus the increment parameter value. When control transfers out of a DO loop, the DO variable retains its most recent value.

You can enter a DO loop only through its DO statement, unless you are reentering a loop from the extended range of that loop. (See Extended Range DO Loops in this chapter.)

Example:

```
   DO 10 J = 1, 11, 3
   IF(A(J) .LE. A(J + 1))ITEMP=A(J)
10 A(J) = A(J + 1)
   PRINT 100, A
```

The statements following DO, up to and including statement 10, are executed four times (J will equal 1, 4, 7, and 10). The PRINT statement is then executed. When the loop is complete, J equals 13.

Example:

```
   DO 10 I = 5, 1, - 1
10 IF(X .GT. B(I))B(I)=0.0
   PRINT 500, B
```

This example illustrates the use of a negative increment parameter. The loop compares X with elements 5, 4, 3, 2, and 1, respectively, of array B. After the DO loop has completed, array B is printed. The DO variable I has a value of zero when the DO loop has completed.

Example:

```
    DO 20 I = 1, 200
    IF(I .GE. IVAR) GO TO 10
20  A(I) = 0.0
10  PRINT 100, A
```

A legal exit from the DO loop is made when the value of the DO variable I is equal to IVAR. For instance, if IVAR=30 and I=30, the branch to the statement labeled 10 is executed and I retains the value 30. If, however, IVAR is greater than 200, the statements following the IF statement are executed sequentially and I will contain the value 201.

For Better Performance

DO loops can be vectorized to result in faster execution of your program. To specify vectorization of DO loops, specify VECTORIZATION_LEVEL= HIGH on the VECTOR_FORTRAN command. For more information about vectorization, see chapter 13, Vectorization.

Nested DO Loops

When a DO loop entirely contains another DO loop, the inner loop is called a nested DO loop. The range of a DO statement can include other DO statements providing each inner DO statement and its final statement are within the containing DO loops.

The final statement of an inner DO loop must be either the same as the final statement of any containing DO loop or must occur before it. If more than one DO loop has the same final statement, a transfer to that statement can be made only from within the range (or extended range) of the innermost DO.

The following examples show some possible DO loop nests. Loops can be completely nested or can share a final statement.

Example 1

```
      DO 100 L = 2,LIMIT
      .
      .
      .
      DO 10 J = 1,10
      .
      .
      .
   10 CONTINUE
      .
      .
      .
      DO 20 K = K1,K2
      .
      .
      .
   20 CONTINUE
      .
      .
      .
  100 CONTINUE
```

Example 2

```
      DO 1 I = 1,10,2
      .
      .
      .
      DO 2 J = 1,5
      .
      .
      .
      DO 3 K = 2,8
      .
      .
      .
    3 CONTINUE
      .
      .
      .
    2 CONTINUE
      .
      .
      .
      DO 4 L = 1,3
      .
      .
      .
    4 CONTINUE
      .
      .
      .
    1 CONTINUE
```

Example 3

```
      DO 5 I = 1,5
      DO 5 J = 1,10
      DO 5 K = J,15
      .
      .
      .
    5 A = B * C
```

The following example illustrates a nested DO loop:

```
    DO 200 I = 1, 10
    A(I) = A(I) + 1.0
        DO 100 J = 1, 20
100     IF(A(I) .GE. B(J))A(I)=0.0
200 CONTINUE
```

The outer loop is executed 10 times. On each pass through the outer loop, the inner loop is executed 20 times, Thus, the inner loop is executed a total of 200 times. To improve clarity, the inner loop is indented.

Example:

```
    DIMENSION A(3, 4, 5)
    DO 10 I = 1, 5
        DO 10 J = 1, 4
            DO 10 K = 1, 3
10          A(K, J, I) = 0.0
```

All elements of array A are set to zero.

A DO loop can be initially entered only through the DO statement. However, a loop can be reentered from its extended range (see Extended Range DO Loops in this chapter).

When you use an IF or GO TO statement to bypass one or more inner loops, the bypassed loops each require a final statement. For example,

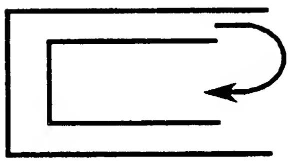
```
    DO 10 I = 1, 100
    IF(I .GT. IVAR) GO TO 10
        DO 20 J = 1, 10
          :
20      CONTINUE
10  CONTINUE
```

In this example, the inner and outer loops cannot share a final statement.

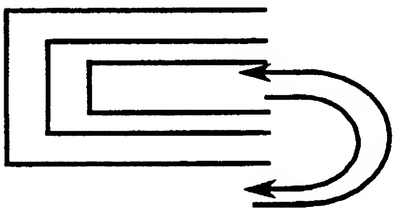
You cannot transfer from an outer DO loop into an inner DO loop unless the transfer is from the extended range of the inner loop. However, you can transfer from an inner DO into outer DO because such a transfer is still within the range of the outer DO loop. A transfer back into an inner or outer DO loop from the extended range of the inner loop is allowed (see Extended Range DO Loops in this chapter).

Subprograms can be called from within a DO loop.

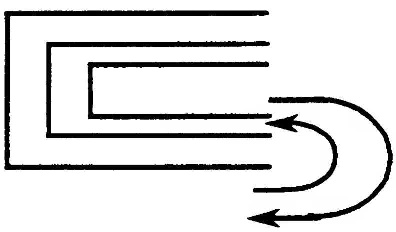
Legal and illegal transfers within a nest of DO loops are shown in the following illustration:



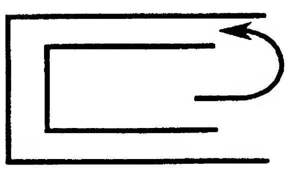
Illegal; you cannot transfer from an outer loop to an inner loop unless the transfer is from the extended range of the inner loop.



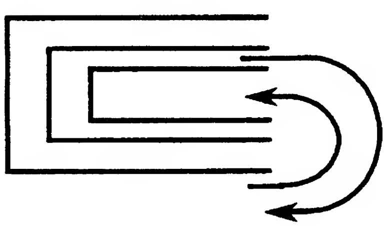
Legal for NOS/VE FORTRAN only; you can transfer back into an inner or outer loop from the extended range of the inner loop.



Legal for NOS/VE FORTRAN only; you can transfer back into an inner or outer loop from the extended range of the inner loop.



Legal; you can transfer from an inner loop to an outer loop.



Illegal; you cannot transfer back into an inner loop from the extended range of an outer loop.

Control Data Extension

Extended Range DO Loops

Under certain conditions, you can transfer control out of a loop and can subsequently transfer back into the loop. The statements executed outside the loop constitute the extended range of the loop. Control can transfer from the extended range back into a loop only if the following conditions are satisfied:

The DO variable is not redefined outside the loop.

The program unit containing the loop has not been exited by a RETURN, STOP, or END statement.

The iteration count of the loop is nonzero.

If any of these conditions are not satisfied, the loop cannot be reentered except through its DO statement.

When DO loops are nested, an inner loop can be exited to the extended range. Subsequent reentry to the inner, or outer, loop can occur subject to the preceding restrictions.

End of Control Data Extension

CONTINUE Statement

The CONTINUE statement is an executable statement that performs no operation. This statement has the form:

CONTINUE

CONTINUE is an executable statement that can be placed anywhere in the executable statement portion of a source program without affecting the sequence of execution. The CONTINUE statement is often used as the last statement of a DO loop. It can end a DO loop when a GO TO or arithmetic IF would normally be the last statement of the loop. For example, the following DO loop is illegal because it ends with an arithmetic IF statement:

```
4    DO 5 I = 1, 100                     }
        A(I) = A(I) + DELT               }        Illegal Do loop
5    IF(I - J) 4, 6, 4                   }
```

This loop is legal and executes correctly if it is changed to the following:

```
      DO 5 I = 1, 100
         A(I) = A(I) + DELT
         IF(I - J) 5, 6, 5
   5  CONTINUE
```

If a CONTINUE statement does not have a label, an informational diagnostic is issued.

PAUSE Statement

The PAUSE statement temporarily stops execution of your program. This statement has the form:

PAUSE *n*

n

Optional; a character constant of 1 through 70 characters or a string of 1 through 5 decimal digits.

When a PAUSE statement is executed in a batch program, execution stops and the message PAUSE *n* is displayed on the system operator console. The operator can continue or terminate the program with an entry from the console.

In an interactive program, execution stops and the message PAUSE *n* is displayed at the terminal. You can terminate the program from the terminal by entering the break sequence defined for the terminal. Any other entry causes execution to continue.

Examples:

```
PAUSE 'EXAMPLE TWO'

PAUSE 45321
```

STOP Statement

The STOP statement ends program execution. This statement has the form:

STOP *n*

n

Character constant of 1 through 70 characters or a string of 1 through 5 decimal digits.

When a STOP statement is encountered during execution, the message STOP *n* is recorded in the job log file, the program terminates, and control returns to the operating system. If you omit *n*, no message is displayed.

You can display the job log file with the NOS/VE DISPLAY_LOG_FILE command as described in the NOS/VE Commands and Functions manual.

A program unit can contain more than one STOP statement.

Examples:

```
STOP
```

```
STOP 'PROGRAM HAS ENDED'
```

END Statement

The END statement indicates the end of the program unit to the FORTRAN compiler. This statement has the form:

END

Every program unit must have an END statement as the last statement.

The END statement can be labeled. If control flows into or branches to an END statement in a main program, execution stops. If control flows into or branches to an END statement in a function or subroutine, action is the same as if a RETURN statement were encountered.

Example:

```
      GO TO 15
        :
   15 END
```

An END statement cannot be continued; it must be completely contained on an initial line. A line following an END statement is considered to be the first line of the next program unit, even if it has a continuation character in position 6.

Input/Output 7

Records 7-1

Files 7-2
File References 7-2
File Attributes 7-3
Default File Attributes 7-3
Changing File Attributes 7-5
File Access Methods 7-6
Sequential Access 7-6
Random Access 7-6
Segment Access 7-7
Opening and Closing Files 7-7

Input/Output Units 7-8
Standard Input/Output Units 7-9
Printer Control Character 7-10

Input/Output Statement Specifiers 7-11

Input/Output Lists 7-13
Implied DO List in Input/Output List 7-15
Nested Implied DO Lists 7-16

Formatted Input/Output 7-18
Formatted READ 7-18
Formatted WRITE 7-20
Formatted PRINT 7-21
Formatted PUNCH 7-21
Format Specification 7-22
FORMAT Statement 7-23
Character Format Specification 7-24
Arithmetic or Boolean Format Specification 7-25
Runtime Format Specification 7-25
Edit Descriptors 7-26
Repeatable Edit Descriptors 7-29
A Descriptor for Character List Items 7-31
Input 7-31
Output 7-32
A Descriptor for Noncharacter List Items 7-33
D Descriptor 7-34
Input 7-34
Output 7-34
E Descriptor 7-35
Input 7-35
Output 7-37
F Descriptor 7-39
Input 7-39
Output 7-39
G Descriptor 7-41
Input 7-41
Output 7-41
I Descriptor 7-42
Input 7-42
Output 7-42

L Descriptor 7-44
Input 7-44
Output 7-44
O Descriptor 7-45
Input 7-45
Output 7-46
R Descriptor 7-47
Z Descriptor 7-48
Input 7-48
Output 7-48
Nonrepeatable Edit Descriptors 7-50
P Descriptor 7-50
Input 7-50
Output 7-51
BN and BZ Descriptors 7-52
S, SP, and SS Descriptors 7-52
H Descriptor 7-53
Quote and Apostrophe Descriptors 7-54
X Descriptor 7-55
T, TL, and TR Descriptors 7-56
Slash (End-of-Record) Descriptor 7-58
Colon (:) Descriptor 7-58

Unformatted Input/Output 7-59
Unformatted WRITE 7-59
Unformatted READ 7-60

List Directed Input/Output 7-61
List Directed READ 7-61
List Directed WRITE, PRINT, and PUNCH 7-64

Namelist Input/Output 7-65
NAMELIST Statement 7-65
Namelist READ 7-66
Namelist WRITE, PRINT, and PUNCH 7-69
Arrays in Namelist 7-70

Buffer Input/Output 7-71
BUFFER IN 7-72
BUFFER OUT 7-73

Mass Storage Input/Output 7-74
Random Files 7-74
Index Key Types 7-75
Master Index 7-75
Subindex 7-75
Multilevel File Indexing 7-76
OPENMS 7-77
WRITMS 7-78
READMS 7-80
CLOSMS 7-80
STINDX 7-81

Direct Access Files 7-83
Creating a Direct Access File 7-84
Direct Access File Examples 7-84
Direct Access Record Length Calculation 7-85

Internal Input/Output 7-86
Standard Internal Files 7-86
Internal WRITE 7-87
Internal READ 7-88
Extended Internal Files 7-89
ENCODE 7-90
DECODE 7-91

Segment Access Files 7-93

Input/Output-Related Statements and Routines 7-95
File Status Statements 7-95
OPEN 7-95
CLOSE 7-100
INQUIRE 7-102
File Positioning Statements 7-108
REWIND 7-108
BACKSPACE 7-109
ENDFILE 7-110
Input/Output Status Checking Functions 7-111
UNIT 7-111
EOF 7-112
IOCHEC 7-113
LENGTH and LENGTHX 7-113
LENGTHB 7-114
Internal Data Transfer Routines 7-115
MOVLEV 7-115
MOVLCH 7-116
File Connection Routines 7-117
CONNEC 7-117
DISCON 7-118
Formatted Data Mismatch 7-118
IGNFDM 7-118

FORTRAN Fast I/O 7-119
Open Sharing 7-119
Files Shared With Another Task or Another Language Subprogram 7-119
Connected Files 7-120
Programs With CALL SCLCMD Statements 7-120
Fast I/O for Individual Files 7-120
How to Turn Off Fast I/O 7-121

Input/Output 7

The FORTRAN input/output operations involve reading records from files and writing records to files. This chapter describes nine types of input/output statements and input/output related routines. The nine types of input/output statements are:

- Formatted
- Unformatted
- List directed
- Namelist
- Buffer
- Mass storage
- Direct access
- Internal
- Segment access

This chapter does not describe the FORTRAN keyed-file interface. It is described in the FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces manual.

The following sections present some NOS/VE terms and concepts you should know before using the FORTRAN input/output statements.

NOTE

Mixing types of operations on the same file can destroy file integrity. In particular, files processed by mass storage or keyed-file subroutines should be processed only by those subroutines.

Records

A record is generally considered to be the amount of data transferred by a single input or output operation. Execution of any input or output statement transfers at least one record. Formatted input/output statements can transfer more than one record.

A record is a contiguous group of bytes within a file; it is read or written as a single unit. The record types used in FORTRAN are:

V Variable length

F Fixed length

The sequential READ and WRITE statements, namelist I/O statements, list directed I/O statements, and buffer I/O statements process sequential files with V type records. The record type can be changed by a SET_FILE_ATTRIBUTE command before execution.

Direct access input/output statements process byte addressable files with F type records. F is the only record type permitted for direct access input/output.

Files

External files reside on an external storage device, such as a disk. Internal files reside in memory. The term file, as used in this manual, refers to an external file. Internal files are described in this chapter under Internal Input/Output.

An external file is a collection of data stored in records. A file begins at its beginning-of-information (BOI) and ends at its end-of-information (EOI).

A file with V type records can contain one or more partition boundaries. Partition boundaries are not allowed on files with F type records. A partition boundary and the end-of-information are recognized as the end-of-file by the FORTRAN input/output statements. The ENDFILE statement writes a partition boundary. Throughout this chapter, the term end-of-file means either end-of-partition or end-of-information.

When an end-of-partition is encountered during a read, the ERR= specifier and EOF function return end-of-file status. If the end-of-partition does not coincide with end-of-information, you can continue reading the same file until the end-of-information is encountered.

For F and U type records, the EOF and UNIT functions return end-of-file status only at end-of-information.

For every formatted, list directed, namelist, or unformatted READ, you should check for end-of-file by using the END= or IOSTAT= specifier in the READ statement. If an end-of-file is encountered and a test is not included, the program terminates with a fatal error.

File References

A FORTRAN program can reference a NOS/VE file by its file path and optionally how the file is positioned prior to use. Examples of file references are:

```
$USER.TESTS.FILEA
:NVE.PAT.EXAMPLES.FTN_PROG.3.$EOI
$LOCAL.TEST
.ACCOUNT.CAT1.FILE2
```

Files referenced in a FORTRAN program without a file path are assumed to be in the working catalog unless the file name specifies a standard system file. Standard system files, such as $INPUT or INPUT, are assumed to be in the $LOCAL catalog. Files that are implicitly created by a FORTRAN program using a file name without a file path are created in the working catalog. For more information about NOS/VE file references, see the NOS/VE System Usage manual.

Unit names used in PROGRAM statements to associate files with units, and unit names derived from unit identifiers specified in boolean format, cannot be NOS/VE file references. They must be 7 or less characters in length.

File Attributes

Every NOS/VE file has associated with it a set of file attributes. These attributes completely describe the structure and processing limitations of the file. The file attributes are stored in an internal table that is created and maintained by NOS/VE. These attributes include file organization, record type, record length, and many others.

Files read and written by the FORTRAN input/output statements are provided with default values for the file attributes, and you usually do not need to be concerned with the file attributes or with the internal operations of NOS/VE. (An exception to this the keyed-file interface subprograms, described in the FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces manual.)

Some values are permanent for the life of the file; others can be overridden by a SET_FILE_ATTRIBUTE or CHANGE_FILE_ATTRIBUTE command, or a PROGRAM or OPEN statement executed before the first time the file is opened. To specify the attributes of a file, see Default File Attributes or Changing File Attributes in the next sections.

Default File Attributes

FORTRAN sets certain file attributes depending on the nature of the input/output operation and its associated file structure. Most attributes are permanent for the life of a file. After a file is created (that is, after the file is opened for the first time), the permanent attributes cannot be changed.

The file attributes for the various types of FORTRAN input/output are shown in table 7-1. The attributes which can be overridden by a SET_FILE_ATTRIBUTE command or CHANGE_FILE_ATTRIBUTE command before creating the file are indicated by a 1 subscript; those attributes which can be overridden before opening the file are indicated by a 2 subscript.

Files connected to $INPUT or $OUTPUT retain the attributes of $INPUT or $OUTPUT regardless of the SET_FILE_ATTRIBUTE and CHANGE_FILE_ATTRIBUTE command specifications.

Table 7-1. FORTRAN Defaults for File Attributes

File Attribute	Formatted Sequential I/O	Unformatted Sequential I/O	Buffer I/O	Mass Storage I/O	Direct Access I/O
MAXIMUM_ RECORD_ LENGTH	RECL= in OPEN statement [3]	RECL= in OPEN statement[1,3]	n/a	n/a	RECL= in OPEN statement
OPEN_POSITION	$BOI[2]	$BOI[2]	$BOI[2]	n/a	n/a
ACCESS_MODE	R/W/A/M[2]	R/W/A/M[2]	R/W/A/M[2]	R/W/A/M[2]	R/W/A/M[2]
FILE_ ORGANIZATION	SQ	SQ	SQ	BA	BA
RECORD_TYPE	V[1]	V[1]	V[1]	U	F
PADDING_ CHARACTER	n/a	n/a	n/a	n/a	blank[1]
PAGE_WIDTH	132 characters (nonconnected file)[1] 72 characters (connected file)[1]	n/a	n/a	n/a	n/a

[1] Can be overridden by SETFA command prior to file creation

[2] Can be overridden by SETFA command prior to any open

[3] If not specified, the default file length is 65535.

n/a = Not applicable to this mode of input/output

$BOI = Beginning of information

R/W/A/M = READ/WRITE/APPEND/MODIFY

SQ = Sequential

BA = Byte-addressable

V = Variable-length

F = Fixed-length

U = Undefined

Changing File Attributes

You can change file attributes by using the SET_FILE_ATTRIBUTE command. The SET_FILE_ATTRIBUTE command changes file attributes from those compiled into a program, and consequently, default FORTRAN input/output processing. In particular, this command enables you to read or create a file with attributes that are different from those supplied by default.

The file attributes specified on a SET_FILE_ATTRIBUTE command are established when a file is created (that is, the first time it is opened).

Only some of the parameters on the SET_FILE_ATTRIBUTE command apply to FORTRAN input/output. See the NOS/VE Commands and Functions manual for a complete description of the SET_FILE_ATTRIBUTE command. The format of the SET_FILE_ATTRIBUTE command and applicable parameters is:

SET_FILE_ATTRIBUTE or
SETFA
  FILE=file
  ACCESS_MODE=list of keywords
  FILE_CONTENTS=keyword
  FILE_ORGANIZATION=keyword
  FILE_STRUCTURE=keyword
  MAXIMUM_RECORD_LENGTH=integer
  OPEN_POSITION=keyword
  PADDING_CHARACTER=character
  PAGE_WIDTH=integer
  RECORD_TYPE=keyword

This format shows only those parameters which are applicable to the FORTRAN files described in this chapter. Refer to the NOS/VE Commands and Functions manual for a complete description of the SET_FILE_ATTRIBUTE command for sequential files.

To honor carriage control, the FILE_CONTENTS attribute must be LIST (to match the OUTPUT attribute)

Example:

```
PROGRAM ABC
OPEN (FILE='AFILE', UNIT=1)
WRITE (1,100) A, B, C
  ⋮
```

This program opens and writes a file named AFILE.

The following SET_FILE_ATTRIBUTE command, specified before the program is executed, overrides the default maximum record length of 65535 characters:

```
/set_file_attribute file=afile maximum_record_length=100
```

A MAXIMUM_RECORD_LENGTH specification in a SETFA command prior to program execution takes precedence over a record length specification in an OPEN or PROGRAM statement. For direct access files, the MAXIMUM_RECORD_LENGTH parameter cannot specify a different record length than the OPEN statement.

File Access Methods

FORTRAN provides three methods of accessing a file: sequential, random, and segment. Sequential and random access methods are for accessing records in a file. Segment access is for accessing data associated with a common block.

Sequential Access

In sequential access files, records are written in sequential order and can only be read in the same order in which they were written. You can access a sequential record only by reading sequentially until the desired record is found.

Sequential files are read and written by FORTRAN READ and WRITE statements, BUFFER IN and BUFFER OUT statements, and PRINT and PUNCH statements. You can perform formatted, unformatted, list directed, buffer, and namelist input/output on sequential files.

To create a sequential access file or reference an existing one, you can specify the ACCESS='SEQUENTIAL' specifier on the OPEN statement for the file or omit this specifier. By default, the file is assumed to be a sequential access file.

Random Access

Random access files provide a quicker method of access to a specific record than sequential access files. Records on a random access file can be read or written in any order by specifying a record key or a record number. Random access files eliminate the need for sequentially searching a file for a particular record.

FORTRAN provides three ways of performing random access input/output:

- The first way is to use the direct access file capability. Direct access files are processed by standard FORTRAN READ and WRITE statements. To create a direct access file, or to reference an existing one, you must specify ACCESS='DIRECT' on the OPEN statement for the file. You can use direct access files for formatted or unformatted input/output. You cannot use direct access files for namelist, list directed, buffer, or internal input/output.

- A second method of random access is provided by the mass storage subroutines. These files, often referred to as mass storage files, must be processed by the mass storage subroutines; they cannot be processed by READ and WRITE statements. The mass storage subroutines are described in this chapter under Mass Storage Input/Output.

- A third method of random access is provided by the keyed-file interface subprograms. These subprograms enable you to perform operations on files having indexed-sequential or direct access organization. Records on keyed-file or direct access files can be accessed directly by record key. These subprograms are described in the FORTRAN for NOS/VE Keyed-File and Sort/Merge Interfaces manual.

If you decide to use one of the random access methods, only the FORTRAN direct access capability is an ANSI standard feature, whereas the mass storage and keyed-file interface capabilities are Control Data extensions and inhibit program portability. Also, direct access files have fixed length records, while both mass storage and file interface allow variable length records. The keyed-file interface capability offers the most flexibility.

Segment Access

Segment access files allow fast and efficient access to large blocks of data. These files are associated, or mapped, with a named common block. Values are accessed and modified through FORTRAN assignment statements rather than input/output statements. This association is accomplished by using the C$ SEGFILE directive and by using the common block name as the UNIT specifier in the OPEN, CLOSE, and INQUIRE statements.

NOTE

When F type records are being written to a sequential file, and an attempt is made to write more characters to a record than the record length allows, the record is truncated and no warning message is issued. The only indication that truncation occurred is given when the truncated record is subsequently read.

Opening and Closing Files

Before you can read or write to a new or existing file in a FORTRAN program, the file must be opened. The opening process prepares the file for input/output and establishes certain file properties, such as unit/file association, record length, access method, and so forth.

You can open a file explicitly by declaring it in an OPEN statement. (Mass storage files require an OPENMS call). Alternatively, for sequential files only, you can simply reference a unit in an input/output statement and the file associated with that unit is automatically opened with default values provided for the various file properties.

After you have finished reading or writing a file, you can close the file by specifying a CLOSE statement (CLOSMS for mass storage files). The CLOSE statement performs various file completion operations. If you do not explicitly close a file, it is automatically closed when the program terminates.

Input/Output Units

All files are referenced in FORTRAN input/output statements through an associated unit. For example, the statement

```
READ(2, 100) A, B
```

specifies unit number 2. The read operation is performed on the file associated with unit 2.

You can associate a file with a unit through parameters on the OPEN statement or with a CREATE_FILE_CONNECTION command prior to execution. If you reference a unit number that has not been associated with a file through an OPEN statement, NOS/VE command, or PROGRAM statement, the file name used is TAPE*n* (in the working catalog), where *n* is the unit number you specified. For example, if a program contains the statement

```
READ(UNIT=9, FMT=150), X, Y
```

and unit 9 is not associated with a file name on an OPEN or PROGRAM statement or NOS/VE command, a file named TAPE9 is read.

You can specify unit names in input/output statements or the PROGRAM statement using the boolean L format. (This capability is provided mainly for compatibility with previous systems.) In this case, the file name is the same as the unit name. File names specified in this manner are limited to seven characters. For example,

```
READ(L"AFILE", 100) A, B
```

reads from a file named AFILE.

A unit specification of the form L"TAPEn" is the same as specifying unit number n. That is, the file associated with unit n is used or, if no file is associated with unit n, the file name used is TAPEn. For example, the following statements are equivalent:

```
READ(UNIT=L"TAPE1", FMT=5) AV, BV

READ(UNIT=1, FMT=5) AV, BV
```

Both statements read from file TAPE1 (assuming unit 1 has not been associated with a file name in an OPEN or PROGRAM statement or NOS/VE command.)

You associate a segment access file with a common block name by using the common block name (including slashes) as the UNIT specifier. For example,

```
      COMMON /CHBLK/CH(1000)
C$    SEGFILE(/CHBLK/)
      OPEN(UNIT=/CHBLK/, FILE='SEG1')
```

These statements open a segment access file named SEG1 in the working catalog. SEG1 is mapped to the common block CHBLK.

Standard Input/Output Units

FORTRAN provides several standard units for use in input/output statements. If you specify an asterisk for the unit in an input/output statement, or if you omit the list of input/output specifiers entirely, the unit defaults to one of the standard units. The standard units have a default association with a standard system file. The system files do not contain data, but are connected to physical files that contain data. The following table shows the standard units and the default files associated with them:

Standard Unit	Standard System File	Physical File	Description
INPUT	$INPUT	INPUT	In an interactive job, data is read from the terminal. To terminate data entry, enter *EOI (must be uppercase) immediately after the prompt. In a batch job, INPUT is connected to $NULL.
OUTPUT	$OUTPUT	OUTPUT	In an interactive job, data is written to the terminal. In a batch job, data is printed at job termination.
PUNCH	PUNCH	-	Data is written to file PUNCH.
-	$NULL	-	Data written to file $NULL is discarded.

Standard system files cannot be redefined (that is, opened with STATUS='NEW' on the OPEN statement), but they can be reconnected to different physical files.

Example:

```
      PROGRAM S
      READ(*, 100) I,J
  100 FORMAT(2I3)
```

The READ statement reads from $INPUT (which is connected to the terminal in an interactive job). You can change the connection to read from another file as follows:

```
CREATE_FILE_CONNECTION $INPUT MYFILE
```

The above program now reads from file MYFILE.

Standard system files reside in the $LOCAL catalog. Other files used by a FORTRAN program are assumed to be in the working catalog.

For Better Performance

Input/output operations now execute faster due to improved internal processing. The benefits are most noticed with buffered, direct-access, and sequential input/output. The improved execution can cause different behavior during input/output processing. See Fast I/O in this chapter for more information.

Printer Control Character

The first character of a printer output record is used for printer control and is not printed. It appears in other forms of output as data. For lines listed at a terminal, the FILE_CONTENTS parameter on the SET_FILE_ATTRIBUTE command allows you to specify whether printer control characters are to be recognized or disregarded. The SET_FILE_ATTRIBUTE command is described under Changing File Attributes in this chapter.

The printer control characters are shown in table 7-2.

Table 7-2. Printer Control Characters

Character	Action
Space	Space vertically one line, then print.
0	Space vertically two lines, then print.
1	Eject to the first line of the next page before printing.
+	No advance before printing; allows overprinting.
-	Space vertically three lines, then print.
Any other character	Determined by the operating system.

Printer control characters are required at the beginning of every record to be printed, including new records introduced by means of a slash. Null records, such as those produced by successive slashes, do not require printer control characters. Printer control characters can be generated by any means.

For output directed to any device other than the line printer or terminal, printer control characters are not required.

Record length on print files should not exceed the length of a printer line (usually 132 or 137 characters). The default maximum record length is 65535 characters. The first character is always used for printer control and is not printed. The second character appears in the first print position.

Print files that are not connected to $OUTPUT must have a FILE_CONTENTS attribute of LIST (for batch) or LEGIBLE (for interactive) to print correctly. Following are some examples of FORMAT statements in which the first character of each record is a printer control character:

```
10    FORMAT (1H0 , F7.3, I2, G12.6)

20    FORMAT ('  ', I5, 'RESULT=', F8.4)

30    FORMAT ('1', I4, 2 (F7.3))

40    FORMAT (1X, I4, G16.8)
```

Input/Output Statement Specifiers

Specifiers are used in input/output statements to select various processing options. The specifiers described below apply to the description of input/output statements throughout this chapter. The input/output statement specifiers are:

UNIT=u

Specifies the FORTRAN unit or internal file to be used. The unit name is derived from the unit identifier **u**, which can be one of the following:

An asterisk implying unit INPUT in a READ statement and unit OUTPUT in a WRITE statement. The default file for unit INPUT is $INPUT; the default file for unit OUTPUT is $OUTPUT.

The name of a character variable, array, array element, or substring identifying an internal file. An array name must not specify an assumed-shape array.

The name of a common block (including slashes) to associate the common block with a segment access file.

An integer or boolean expression having the following characteristics:

INT(u) has a value in the range 0 through 999. The compiler associates these numbers with unit names of the form TAPEu.

or

BOOL(u) is an ASCII coded name in boolean L format (left-justified with binary zero fill). This is the unit name. If this name is of the form TAPEk, where k is an integer in the range 0 through 999 with no leading zero, it is equivalent to the integer k for the purpose of identifying external units. A valid unit name consists of one through seven letters or digits beginning with a letter. (Uppercase and lowercase letters are equivalent.)

The characters UNIT= can be omitted, in which case **u** must be the first item in the list of specifiers.

File names default to the unit name unless a different file name has been specified on the PROGRAM or OPEN statement or execution command. Files that default to the unit names are assumed to be in the working catalog.

When unit is an integer expression and it is passed to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be an 8-byte integer expression.

FMT=fn

Specifies a format to be used for formatted input/output; **fn** can be one of the following:

Statement label of a FORMAT statement in the program unit containing the input/output statement.

Character array or expression containing the format specification. An array name must not specify an assumed-shape array.

An arithmetic or boolean array containing the format specification. An array name must not specify an assumed-shape array. Integer arrays must be 8-byte integer arrays.

An integer variable that has been assigned the statement number of a FORMAT statement by an ASSIGN statement.

An asterisk, indicating list directed I/O.

A namelist group name.

The characters FMT= can be omitted, in which case the format specifier must be the second item in the list of specifiers, and the first item must be the unit specifier without the characters UNIT=.

REC=rn

Specifies the number of the record to be read or written in the file; must be a positive nonzero integer. Valid only if the unit is open for direct access.

END=sl

Specifies the label of an executable statement to which control transfers when an end-of-file is encountered during an input operation.

ERR=sl

Specifies the label of an executable statement to which control transfers if an error condition is encountered during input/output processing.

IOSTAT=ios

Specifies an 8-byte integer variable or array element into which one of the following values is returned after the input/output operation is complete:

Value Returned	Status of Input/Output Operation
< 0	End-of-file encountered
= 0	Operation completed normally
> 0	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the 2-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors. A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'.

Errors are listed by error number and condition name in the NOS/VE Diagnostic Messages, which provides descriptions and suggested action for the errors. To determine the condition name of another product's condition code, use the INTCOND function or the NOS/VE function $CONDITION_NAME with the returned condition code. The INTCOND function is described in chapter 10. See the NOS/VE System Usage manual for more information.

iolist

Input/output list specifying items to be transmitted (described in the next section under Input/Output Lists).

Input/Output Lists

The list portion of an input/output statement specifies the items to be read or written and the order they are read or written. The input/output list can contain any number of items. List items are read or written sequentially from left to right.

If no list appears on input, one or more records are skipped. If no list appears on formatted output, only information completely contained within the FORMAT statement, such as character strings, is transmitted. If no list appears on unformatted output, a null (empty) record is transmitted.

A list item can be a variable name, an array name, an array section reference, an array element name, a character substring name, or an implied DO list. On output, a list item can also be a character, boolean, logical, or arithmetic expression. An array name without subscripts or an array section reference in an input/output list specifies the entire array or array section in the order in which it is stored. The entire array or array section is read or written.

No expression in an input/output list can reference a function if such reference would cause any input/output operations to be executed or would cause the value of any element of the input/output statement to be changed. List items are separated by commas.

You cannot use assumed-size array names in input/output lists. However, you can use assumed-size array section or element references.

The following input/output statements show typical input/output lists:

```
READ(2, 100) A, B, C, D

READ(3, 200) A, B, C(I), D(3, 4), E(I, J, 7), H

READ(4, 101) J, A(J), I, B(I, J)

WRITE(2, 202) DELTA

WRITE(4, 102) DELTA (5 * J + 2, 5 * I - 3, 5 * K), C, D(I + 7)

WRITE(4, 100) V(1:100, 3)
```

On formatted input or output, the input/output list is scanned and each item in the list is paired with the edit descriptor provided by the FORMAT statement. After one item has been input or output, the next edit descriptor is taken together with the next element of the list, and so on until the end of the list. For example, the statements

```
      READ(5, 20) L, M, N
   20 FORMAT(I3, I2, I7)
```

read the input record

```
100223456712
```

as follows:

100 is read into the variable L under the specification I3.

22 is read into the variable M under the specification I2.

3456712 is read into N under the specification I7.

On unformatted input or output, the list items are transmitted between memory and the storage device with no formatting.

Implied DO List in Input/Output List

An implied DO list in an input/output list has the form:

(dlist, var=expl, exp2, *exp3*)

dlist

A list of input/output items

var

Integer, real, or double precision variable called the DO variable.

expl

Initial parameter

exp2

Terminal parameter

var

Increment parameter. The increment parameter must not equal zero. If *exp3* is omitted, the increment is one (the preceding comma must also be omitted.

In an input statement, the DO variable must not appear as a list item in **dlist**. The range of an implied DO list is the list of elements in **dlist**.

When an implied DO list appears in an input/output list, the items in dlist are specified once for each iteration of the implied DO, with appropriate substitution of values for any occurrence of the DO variable. For example, all the following statements have the same effect:

```
READ(5, 100) (A(I), I=1, 3)

READ(5, 100) A(1), A(2), A(3)

READ(5, 100) A(1:3)
```

These statements read three records, each containing a value for A.

The DO variable must not be redefined within the range of the implied DO list by a READ statement. For example, the following statement is illegal:

```
READ *, (I, A(I), I=1, 10)
```

Changes to the values of expl, exp2, and exp3 have no effect upon the execution of the implied DO. However, if their values are changed in a READ statement if they are outside the range of the implied DO, the change does have effect. For example, the statement

```
READ 100, K, (A(I), I=1, K)
```

transmits a value to K. That value is then used as the terminal parameter of the implied DO.

You can use an implied DO list to transmit a list item more than one time. For example:

```
WRITE(3, 20) (CAT, DOG, RAT, I=1, 10)
```

This statement writes the sequence CAT, DOG, RAT 10 times each.

A variable cannot be used as a DO variable more than once in the same implied DO nest, but *iolist* items can appear more than once. The value of a DO variable within an implied DO list is defined within that DO list. When the implied DO has completed, the DO variable retains the first value to exceed the upper limit.

Nested Implied DO Lists

Implied DO lists can be nested; that is, the *iolist* in an implied DO list can itself contain implied DO lists. The first (innermost) DO variable varies most frequently, and the last (outermost) DO variable varies least frequently. For example, a nested implied DO with two levels has the form:

((**dlist**, **var1=exp1**, **exp2**, *exp3*), **var2=exp4**, **exp5**, *exp6*)

Nested implied DO lists are executed the same as nested DO loops (see Nested Do Loops in chapter 6, Flow Control).

Example:

```
    DIMENSION VECT(3, 4, 7)
    READ (3, 100) VECT
100 FORMAT (I6)
```

This sequence is equivalent to the following:

```
DIMENSION VECT(3, 4, 7)
READ(3, 100) (((VECT(I, J, K), I=1, 3), J=1, 4), K=1, 7)
```

Example:

```
READ(1, 100) ((E(I, J), J=1, 3), I=1, 3)
```

This statement transmits nine elements into the array E in the order:

E(1, 1)
E(1, 2)
E(1, 3)
E(2, 1)
E(2, 2)
E(2, 3)
E(3, 1)
E(3, 2)
E(3, 3)

Example:

```
     WRITE(2, 200) ((I, E(I, J), J=1, 3), I=1, 3)
200  FORMAT(1X, I1, 9F7.2)
```

This sequence writes the variable I and nine elements from array E in the order:

E(1,1)
E(1,2)
E(1,3)
E(2,1)
E(2,2)
E(2,3)
E(3,1)
E(3,2)
E(3,3)

Example:

```
     READ (5, 100) (VECTOR (I), I=1, 10)
100  FORMAT (F7.2)
```

These statements read data (consisting of one data item per record) into the elements 1 through 10 of the array VECTOR. The following statements have the same effect (but are less efficient):

```
     DO 40 I=1,10
40   READ (5, 100) VECTOR (I)
100  FORMAT (F7.2)
```

In this example, numbers are read, one from each record, into the elements VECTOR(1) through VECTOR(10) of the array VECTOR. The READ statement is encountered each time the DO loop is executed; and a new record is read for each element of the array.

If the format specification F7.2 is changed to 4F7.2, only three records are read by the first example; the second example still reads 10 records. Both examples read 10 values.

Formatted Input/Output

For formatted input/output, a format specification (**fn**) must be present in the input/output statement. The *iolist* specifier is optional. Each formatted input/output statement transfers one or more records. Each record must be zero or more characters in length.

NOTE

For sequential formatted input/output, if you try to read more characters than the record contains, the truncated portion of the record is read as blanks. No message is issued.

The formatted input/output statements described in this section are:

- Formatted READ
- Formatted WRITE
- Formatted PRINT
- Formatted PUNCH

Format specifications are described following the formatted input/output statements.

Formatted READ

The formatted READ statement transmits data from a storage device to internal storage, and converts the data according to a format specification. This statement has the forms:

READ(*UNIT=* **u,** *FMT=* **fn,** *IOSTAT=ios, ERR=sl, END=sl) iolist*

READ fn, *iolist*

The UNIT=, FMT=, IOSTAT=, ERR=, and END= specifiers are described earlier in this chapter under Input/Output Statement Specifiers. The iolist is described under Input/Output Lists.

The first form transmits data from the file associated with unit u to storage locations named in *iolist* according to FORMAT specification fn. The second form performs the same operation for unit INPUT. The number of items in the list and the FORMAT specifications must conform to the record structure on the input unit. If the list is omitted, a record is bypassed. (Slash descriptors in the format specification causes additional records to be bypassed.)

Each execution of a READ statement transmits at least one record. The FORMAT statement determines when a new record is read.

```
    READ (5, 100) (VEC(I), I=1,10)
100 FORMAT (5F7.2)
```

The READ statement reads data (consisting of 5 data items per record) into the first 10 elements of array VEC.

You must specify the END= or IOSTAT= specifier to test for an end-of-file. Do not read a unit after an END= or IOSTAT= specifier has returned an end-of-file condition for that unit. Records following an end-of-partition can be read by using a CLOSE statement followed by an OPEN statement on the file or by using the EOF function.

Example:

```
READ (4, 200) A, B, C
```

This statement reads data from unit 4 into the variables A, B, and C according to the specifications in the format statement labeled 200.

Example:

```
READ 5, X, Y, Z
```

This statement reads data from unit INPUT (unit specifier omitted) to the variables X, Y, and Z according to the specifications in the format statement labeled 5.

Example:

```
READ (2, 100, ERR=16, END=18)
```

This statement reads data from the file associated with unit 2 into variables A and B according to format 100. If an error occurs during the read, control transfers to the statement labeled 16; if an end-of-file is encountered, control transfers to the statement labeled 18.

Example:

```
READ (2, '(2F10.4)') A, B
```

This statement reads data into a and b from the file associated with unit 2 according to the format specification 2F10.4.

Formatted WRITE

The formatted WRITE statement transfers data from the storage locations named in the *iolist* specifier to the file associated with the unit specified by u. This statement has the form:

WRITE (*UNIT=* **u**, *FMT=* **fn**, *ERR=sl, IOSTAT=ios) iolist*

The UNIT=, FMT=, ERR=, and IOSTAT= specifiers are described earlier in this chapter under Input/Output Statement Specifiers. The iolist is described under Input/Output Lists.

The data is converted from internal format to coded format according to the format specification fn.

Each execution of a WRITE statement transmits at least one record. The FORMAT statement determines when a new record is transmitted. If the *iolist* is omitted (and no H, quote, or apostrophe edit descriptors are specified), an empty record is written.

Example:

```
WRITE (UNIT=4, FMT=50) A, B
```

This statement writes data from variables A and B to unit 4 according to the FORMAT statement labeled 50. The keywords UNIT= and FMT= in the unit and format specifiers are optional.

Example:

```
WRITE (*, 12) L, M, S(3)
```

This statement writes data from variables L, M, and S(3) to unit OUTPUT (as indicated by an asterisk in place of the unit specifier) according to the format statement labeled 12.

Example:

```
WRITE (4, 50, ERR=200) A, B
```

This statement is identical to the first example except that if an error occurs during the write, control transfers to statement 200.

Example:

```
WRITE (2, '(3E12.4)') XVAR, YVAR
```

This statement writes variables XVAR and YVAR to the file associated with unit 2 according to the specification 3E12.4. This example shows a format specification included in the WRITE statement.

Formatted PRINT

The PRINT statement transfers information from the storage locations named in the *iolist* specifier to unit OUTPUT according to the specified format. The default association of unit OUTPUT is to file $OUTPUT. This statement has the form:

PRINT fn, *iolist*

The format specification fn is described earlier in this chapter under Input/Output Statement Specifiers. The iolist is described under Input/Output Lists.

Example:

```
PRINT 4, A, B, N
```

This statement transfers data from A, B, and N to unit OUTPUT according to the format statement labeled 4.

Example:

```
PRINT '(3E14.4)', X1, X2, X3
```

This statement transfers data from X1, X2, and X3 to unit OUTPUT according to the format specification 3E14.4.

Control Data Extension

Formatted PUNCH

The PUNCH statement transfers data from the specified storage locations to the unit PUNCH. The default association of unit PUNCH is to file PUNCH. This statement has the form:

PUNCH fn, *iolist*

The format specification fn is described earlier in this chapter under Input/Output Statement Specifiers. The iolist is described under Input/Output Lists.

Example:

```
PUNCH 5, A, B, C, ANSWER
```

This statement writes data from variables A, B, C, and ANSWER according to the FORMAT statement labeled 5.

Example:

```
      PUNCH 30
30    FORMAT (' GOOD MORNING')
```

This statement writes according to the FORMAT statement labeled 30. Since no *iolist* is specified, no data is transferred and converted from variables.

End of Control Data Extension

Format Specification

Format specifications are used in conjunction with formatted input/output statements to read or write strings of ASCII characters. On input, data is received in the specified format and converted to an internal representation. On output, data is converted from an internal representation to the specified format and written as records of ASCII characters.

Format specifications are identified by the FMT=fn specifier on an input/output statement; fn can be one of the following:

- The statement label of a FORMAT statement in the program unit containing the input/output statement. The FORMAT statement is described in the next section.
- A character array or expression containing the format specification. These format specifications are described under Character Format Specification in this chapter.
- An arithmetic or boolean array containing the format specification. These format specifications are described under Arithmetic and Boolean Format Specification in this chapter.
- An integer variable that has been assigned the statement number of a FORMAT statement by an ASSIGN statement. The ASSIGN statement is described in chapter 6, Flow Control.
- A namelist group name. Namelist group names are described under Namelist Input/Output in this chapter.

All of the above format specifications contain edit descriptors to specify how the input or output data is formatted. Edit descriptors are described in this chapter.

Format specifications can also be determined at runtime. For more information, see Runtime Format Specification in this chapter.

FORMAT Statement

The FORMAT statement is a nonexecutable statement which specifies the formatting of data to be read or written with formatted input/output. This statement has the form:

sl FORMAT (item, ..., *item)*

sl

Statement label

item

One of the following:

A nonrepeatable edit descriptor

A repeatable edit descriptor optionally preceded by a repeat count

A list of repeatable or nonrepeatable edit descriptors, enclosed in parentheses, and optionally preceded by a repeat count

The FORMAT statement is used with formatted input and output statements. It can appear anywhere in a program unit after the PROGRAM, FUNCTION, or SUBROUTINE statement. An example of a READ statement and its associated FORMAT statement is as follows:

```
    READ (5, 100) INK, NAME, AREA
100 FORMAT (10X, I4, I2, F7.2)
```

The format specification consists of a sequence of edit descriptors enclosed in parentheses. Spaces are not significant except in H, quote, and apostrophe edit descriptors.

Generally, each item in the *iolist* specifier is associated with a corresponding edit descriptor in the FORMAT statement. The FORMAT statement specifies the external format of the data and the type of conversion to be performed.

The type of conversion should correspond to the type of the variable in the *iolist* specifier. The FORMAT statement specifies the type of conversion for the input data, with no regard to the type of the variable which receives the value when reading is complete. For example, the following statements are incorrect:

```
    INTEGER N          ! These statements
    READ (5, 100) N    ! are
100 FORMAT (F10.2)     ! incorrect
```

These statements are incorrect because an integer variable is being read with a floating point format. However, you can use the IGNFDM subroutine to ignore data type and FORMAT statement mismatches for noncharacter data. The IGNFDM call is described in this chapter under Input/Output Related Statements and Routines.

Character Format Specification

Instead of a statement label, a format specification can also be a character expression or a character array that contains a format specification. The character expression is substituted for the FORMAT statement number in the FMT= specifier of the READ or WRITE statement.

NOTE

The character expression must not involve concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is also declared as a symbolic constant.

The form of these format specifications is the same as for FORMAT statements without the keyword FORMAT. Any character information beyond the terminating parenthesis is ignored. The initial left parenthesis can be preceded by spaces. For example, the sequence:

```
CHARACTER FORM*11
DATA FORM/'(I3,2E14.4)'/
READ (2, FMT=FORM, END=50) N, A, B
```

is equivalent to:

```
    READ (2,FMT=100,END=50) N, A, B
100 FORMAT (I3,2E14.4)
```

The preceding examples can also be expressed as:

```
READ (2,FMT='(I3, 2E14.4)',END=50) N, A, B
```

or

```
CHARACTER FORM*(*)
PARAMETER (FORM='(I3,2E14.4)')
READ (2,FMT=FORM,END=50) N, A, B
```

If a format specification is contained in a character array, the specification can occupy two or more contiguous array elements. Only the array name need be specified in the input/output statement; all information up to the closing parenthesis is considered to be part of the format specification. For example, the statements

```
CHARACTER AR(2)*10
DATA AR/'(10X,2I2,1','0X,F6.2)'/
READ (5, AR) I, J, X
```

read data into I, J, and X according to the format specification contained in the character array elements AR(1) and AR(2). These statements are equivalent to the following statements:

```
    READ (5, 100) I, J, X
100 FORMAT (10X, 2I2, 10X, F6.2)
```

Control Data Extension

Arithmetic or Boolean Format Specification

You can place format specifications in an arithmetic or boolean array. If the array is of type integer or logical, it must be an 8-byte array. The rules for noncharacter format specifications are the same as for character format specifications.

End of Control Data Extension

Runtime Format Specification

Instead of including the format specification in your program, format specifications can be read at runtime. The format can be read under the A edit descriptor specification and stored in a character array, array section, variable, or array element; or it can be included in a DATA statement. Formats can also be generated by the program at runtime. (Runtime format specifications, however, are not interpreted by the compiler, and are therefore slower to execute.)

If you use an array to store a format specification, it can be any type other than character. The format must consist of a list of descriptors and editing characters enclosed in parentheses, but without the keyword FORMAT and the statement label.

The name of the entity containing the specification is used in place of the FORMAT statement number in the associated input/output statement. The name specifies the location of the format information. For example, the input string:

(E7.2,G20.5,F7.4,I3)

can be read and subsequently referenced as follows:

```
CHARACTER F*30
READ (2, '(A)') F
WRITE (3, F) A, B, C, N
```

The preceding example produces the same output as the following statements:

```
    WRITE (3, 10) A, B, C, N
10  FORMAT (E7.2, G20.5, F7.4, I3)
```

A program can create a format specification at runtime. For example, the following statements define a format specification containing a printer control character; if the variable PRINT_FLAG is zero, the printer control character is removed:

```
CHARACTER FMT*9
DATA FMT/'(1X, 3I10)'/
IF (PRINT_FLAG .EQ. 0) FMT (2:4)='
WRITE (2, FMT) I, J, K
```

If PRINT_FLAG is zero, the program produces the same result as

```
WRITE (2, '(3I10)') I, J, K.
```

Edit Descriptors

Format specifications are composed of edit descriptors that specify the data conversions to be performed. Tables 7-3 and 7-4 list the edit descriptors and give a brief description of each. The descriptors listed in table 7-3 can be preceded by an unsigned nonzero decimal integer indicating the number of times the descriptor is to be repeated (as described later in this chapter under Repeatable Edit Descriptors).

Table 7-3. Repeatable Edit Descriptors

Edit Descriptor	Data Type Used With	Description
Ew.d	Numeric	Floating-point with exponent.
Ew.dEe	Numeric	Floating-point with explicitly specified exponent length.
Fw.d	Numeric	Floating-point without exponent.
Dw.d	Numeric	Floating-point with exponent.
Gw.d	Numeric	Floating-point with or without exponent.
Gw.dEe	Numeric	Floating-point with or without exponent (if exponent is present, exponent length is explicitly specified).
Iw	Numeric	Decimal integer.
Iw.m	Numeric	Decimal integer with minimum number of digits.
Lw	Logical	Logical.
A	Character	Character with data-dependent length.
Aw	Character or Boolean	Character or boolean with specified length.
Rw	Boolean	Boolean conversion.
Ow	Boolean	Octal integer.
Ow.m	Boolean	Octal integer with leading zeros and minimum number of digits.
Zw	Boolean	Hexadecimal integer.
Zw.m	Boolean	Hexadecimal with leading zeros and minimum number of digits.

Table 7-4. Nonrepeatable Edit Descriptors

Descriptor	Descriptor Type	Description
SP	Numeric output control	Plus signs (+) produced.
SS		Plus signs (+) suppressed.
S		Plus signs (+) suppressed.
nX	Tabulation control	Position forward.
Tn		Position to column n.
TRn		Position forward n columns.
TLn		Position backward n columns.
nH	Character output	Output character string.
"		Output character string.
'		Output character string.
:	Format control	Terminates format control if no more items in *iolist.*
/	End of record	Indicates end of current input or output record.
kP	Scale factor	Scaling for numeric editing.
BN	Numeric input control	Spaces ignored.
BZ		Spaces treated as zeros.

Symbols representing information that you must supply are as follows:

w Nonzero unsigned integer constant specifying the field width in number of character positions in the external record. This width includes any leading spaces, + or - signs, decimal point, and exponent.

d Unsigned integer constant specifying the number of digits to the right of the decimal point within the field. On output all numbers are rounded to d digits.

e Nonzero unsigned integer constant specifying the number of digits in the exponent; the value of e cannot exceed six.

m Unsigned integer constant specifying the minimum number of digits to be output.

k Integer constant scale factor (used with P descriptor).

n Positive nonzero integer. The meaning of this value depends on the particular edit descriptor.

You must specify the field width w for all edit descriptors except A.

Field separators separate descriptors and groups of descriptors. The format field separators are the slash, the comma, and the colon. (The slash can also specify demarcation of formatted records; the colon terminates format control if no more items are in the *iolist.*)

Leading spaces are not significant in numeric input conversions; other spaces in numeric conversions are ignored unless BLANK='ZERO' is specified for the file on an OPEN statement or a BZ edit descriptor is in effect. You can omit plus signs. An all-spaces field is considered to be zero, except for logical input, where an all-spaces field is considered to have the logical value false, or for character input.

For the E, F, G, and D input conversions, a decimal point in the input field overrides the decimal point specification of the edit descriptor.

The output field is right-justified for all output conversions. If the number of characters produced by the conversion is less than the field width, leading spaces are inserted in the output field unless w.m is specified, in which case leading zeros are produced as necessary. If the number of characters produced by a numeric output conversion exceeds the field width, asterisks are inserted throughout the field.

Complex data items are converted on input or output as two independent floating-point quantities. The format specification for a complex data item uses two edit descriptors. For example, the statements

```
      COMPLEX A
      WRITE (6, 10) A
   10 FORMAT (F7.2, E8.2)
```

write the real part of A according to F7.2 format and the imaginary part according to E8.2 format.

Different types of data can be read by the same FORMAT specification. For example, the statement

```
   10 FORMAT (I5, F15.2)
```

specifies two conversions: the first of type integer, and the second of type real.

The statements:

```
      CHARACTER R*4
      READ (5,15) NO, NONE, INK, A, B, R
   15 FORMAT (3I5, 2F7.2, A4)
```

read three integer values, two real values, and one character string.

Repeatable Edit Descriptors

Certain edit descriptors can be repeated by prefixing the descriptor with a nonzero unsigned integer constant specifying the number of repetitions. The repeatable edit descriptors are A, D, E, F, G, I, L, O, R, and Z. The other edit descriptors cannot be repeated. For example, the following statements are equivalent:

```
100 FORMAT (3I4,2E7.3)

100 FORMAT (I4, I4, I4, E7.3, E7.3)
```

A group of descriptors can be repeated by enclosing the group in parentheses and prefixing it with the repetition factor. If no integer precedes the left parenthesis, the repetition factor is 1. For example, the statement:

```
1   FORMAT (I3, 2(E15.3, F6.1, 2I4))
```

is equivalent to the following specification:

```
1   FORMAT (I3, E15.3, F6.1, I4, I4, E15.3, F6.1, I4, I4)
```

A maximum of nine levels of parentheses is allowed in addition to the parentheses required by the FORMAT statement.

If there are fewer items in the *iolist* specifier than indicated by the format conversions in the format specification, the excess conversions are ignored.

If the total number of items in the *iolist* specifier exceeds the number of format conversions encountered when the final right parenthesis in the FORMAT specification is reached, the converted items are transmitted and a new record is read or written. The format specification is then scanned to the left for a right parenthesis. If none is found, the scan stops when the beginning of the format specification is reached. If a right parenthesis is found, however, the scan continues to the left until the field separator (slash, comma, or colon) which precedes the left parenthesis pairing the right parenthesis is reached. Transmission resumes with formatting proceeding to the right until either the output list is exhausted or the final right parenthesis of the FORMAT statement is encountered. For example, in the sequence

```
    READ (5, 100) I, A, J, B, K, C, L
100 FORMAT (I7, (F12.7, I3))
```

I is input with format I7, A is input with F12.7, and J is input with I3. The format specification is exhausted (the right parenthesis has been reached); a new record is read, and the specification (F12.7,I3) is rescanned. B is input with format F12.7, K with I3, and from a third record, C with F12.7, and L with I3.

A repetition factor can be used to indicate multiple end-of-record slashes; the specification n(/) causes n−1 lines to be skipped on output. For example, the statement

```
2    FORMAT (' VALUES',4(/),' X    Y')
```

causes the following record to be output:

```
VALUES
(blank line)
(blank line)
(blank line)
X Y
```

Editing complex variables always requires two single precision, floating-point (E, F, or G) edit descriptors; the two descriptors can be different. Double precision variables correspond to one floating-point edit descriptor. The D edit descriptor corresponds to exactly one iolist item. A D descriptor is permitted for a complex input item, but the transferred value is truncated to single precision.

Following are descriptions of the repeatable edit descriptors in alphabetical order.

A Descriptor for Character List Items

The A descriptor can be used with an *iolist* specifier item of type character. This descriptor has the forms:

A (Character data only)
Aw

Input

If **w** is less than the length of the *iolist* item, the input quantity is stored left-justified in the item; the remainder of the item is blank-filled. If **w** is greater than the length of the item, the rightmost characters are stored, and the remaining characters are ignored.

If **w** is omitted, the length of the field is equal to the length of the *iolist* item. If **w** is omitted and the *iolist* item is a character array name, then the entire array is read; the length of the field is the length of an array element.

Following are some examples of A input.

Example:

```
    CHARACTER A*9
    READ (5, 100) A
100 FORMAT (A7)
```

Input Record:

```
EXAMPLE
```

In Location A:

```
EXAMPLEΔΔ
```

Example:

```
    CHARACTER B*10
    READ (5, 200) B
200 FORMAT (A13)
```

Input Record:

```
SPECIFICATION
```

In Location B:

```
CIFICATION
```

Example:

```
    CHARACTER Q*8, P*12, R*9
    READ (5, 10) Q, P, R
10  FORMAT (A8, A12, A5)
```

Input Record:

```
THISΔISΔ|ΔANΔEXAMPLEΔI|ΔKNOW
```

(Bars mark input fields as specified by the FORMAT statement.)

In Storage:

```
Q: THISΔISΔ
P: ANΔEXAMPLEΔI
R: ΔKNOWΔΔΔΔ
```

Output

If **w** is less than the length of the *iolist* item, the leftmost characters in the item are output. For example, the statements

```
CHARACTER A*6
A='SAMPLE'
PRINT '(1X, A4)', A
```

print the following:

```
SAMP
```

If **w** is greater than the length of the *iolist* item, the characters are output right-justified in the field, with spaces on the left. For example, if the format specification in the preceding example is changed to (1X,A12), output is:

```
ΔΔΔΔΔΔSAMPLE
```

If **w** is omitted, the length of the output field is the same as the length of the character *iolist* item.

A Descriptor for Noncharacter List Items

The A descriptor, when used for noncharacter *iolist* item, has the form:

Aw

When the A descriptor is used with a noncharacter *iolist* item, the field width specifier, **w**, must appear; **w** characters are converted.

On input, if **w** is less than or equal to 8 (for real, integer, byte, logical, or boolean *iolist* items) or 16 (for double precision or complex *iolist* items), the **w** digits of the input value are converted to character code and stored left-justified in the item with blank fill on the right. If **w** is greater than 8 (for real, integer, byte, logical, or boolean list items) or 16 (for double precision or complex *iolist* items), the rightmost 8 or 16 characters of the input value are converted and stored.

Example:

```
      READ (1,99) X, Y, Z
   99 FORMAT (3A6)
```

Input record:

```
123.4Δ|586.25|Δ1.E04
```

(Bars mark input fields as specified by the FORMAT statement.)

In storage:

```
X:  123.4ΔΔΔ
Y:  586.25ΔΔ
Z:  Δ1.E04ΔΔ
```

A left-justified character string is stored in each of the variables X, Y, and Z.

On output, if **w** is less than or equal to 8 (for real, integer, byte, logical, or boolean *iolist* item) or 16 (for double precision or complex *iolist* item), the internal value is treated as a string of character coded data, and the leftmost **w** characters are written to the output record. If **w** is greater than 8 (for real, integer, byte, logical, or boolean *iolist* item) or 16 (for double precision or complex *iolist* item), the output value is right-justified in the field and preceded by spaces.

D Descriptor

The D descriptor specifies conversion between an internal double precision real number and an external floating-point number written with an exponent. This descriptor has the form:

Dw.d

Input

D editing corresponds to E editing and can be used to input the same forms as E.

The subfields of a D input field are the same as for E input except that you can specify either D or E for the exponent.

Output

Type D conversion is used to output double precision values. D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield. If the value being converted is indefinite, an I is printed in the field; if it is infinite (out-of-range), an R is printed.

The specification Dw.d produces output in the following format:

s.aD±ee

s.a±eee

s

Minus sign if the number is negative, or omitted if the number is positive (subject to control by the S, SS, and SP descriptors)

a

One or more most significant digits

ee

A 2-digit exponent (less than 100)

eee

A 3-digit exponent (100 through 999)

The first form is used for values where the magnitude of the exponent is less than 100; the second form is used for numbers where the magnitude of the exponent is in the range 100 through 999. If the exponent exceeds 999 in magnitude, a field of asterisks is written.

E Descriptor

The E descriptor specifies conversion between an internal real or double precision value and an external number written with an exponent. This descriptor has the forms:

Ew.d
Ew.dEe

Input

An E input field consists of an integer subfield, a fraction subfield, and an exponent subfield.

The integer subfield begins with a + or - sign, a digit, or a space; and it can contain a string of digits. The integer subfield is terminated by a decimal point, E, +, - or the end of the input field.

The fraction subfield begins with a decimal point and terminates with an E, +, - or the end of the input field. It can contain a string of digits.

The exponent subfield can begin with E (or e), +, or -. When it begins with E, the + is optional between E and the string of digits in the subfield. For example, the following are valid equivalent forms for the exponent 3:

```
E+ 03
e 03
e03
E3
+3
```

A nonblank input field must contain an integer subfield or a fraction subfield, or both, and can contain an exponent subfield. An input field cannot contain only an exponent subfield. An input field consisting entirely of spaces is interpreted as zero.

The range, in absolute value, of permissible values is approximately 10**(-1234) through 10**1232. Numbers below the range are treated as zero; numbers above the range are illegal.

Examples of valid subfield combinations are as follows:

+1.6327E-04	Integer-fraction-exponent
-32.7216	Integer-fraction
+328+5	Integer-exponent
.629E-1	Fraction-exponent
+136	Integer only
136	Integer only
.07628431	Fraction only

An exponent subfield alone is not permitted. The width **w** of an E descriptor includes plus or minus signs, digits, decimal point, E, and exponent. If an external decimal point is not provided, **d** acts as a negative power-of-10 scaling factor. The internal representation of the input quantity is given by

i * 10**(-d) * 10**p

where *i* is the integer subfield and *p* is the exponent subfield.

For example, if the specification is E10.8, the input number 3267E+05 is converted and stored as:

3267 * 10**5 * 10**(-8)=3.267.

If an external decimal point is provided in the input number, it overrides a decimal point specified in the E descriptor. If **e** is specified, it has no effect on input.

If the field length specified by **w** in Ew.d is not the same as the length of the field containing the input number, incorrect numbers might be read, converted, and stored. The following example shows a situation where numbers are read incorrectly, converted, and stored; yet there is no immediate indication that an error has occurred:

```
      OPEN (3, BLANK='ZERO')
      READ (3, 20) A, B, C
20    FORMAT (E9.3, E7.2, E10.3)
```

Input Record (three adjacent fields in positions 1 through 24):

```
+6.47E-01|-2.36|+5.321E+02
```

(Bars mark programmer's intended values.)

Numbers actually read:

```
+6.47E-011|2.36+5|.321E+02
```

First, +647E−01 is read, converted and placed in location A. The second specification E7.2 exceeds the width of the second field by two characters. The number −2.36+5 is read instead of −2.36. The specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (−2.36+5) is a legitimate input number (equal to 2.36E+5). Since the second specification incorrectly took 2 digits from the third number, the specification for the third number is now incorrect. The field .321E+02 is read. The OPEN statement specifies that trailing spaces are to be treated as zeros; therefore the number .321E+0200 is read, converted, and stored in location C. Here again, this is a legitimate input number which is converted and stored, even though it is not the number desired.

Following are some additional examples of Ew.d input.

Input Field	Specification	Converted Value	Remarks
+143.26E−03	E11.2	0.14326	All subfields present.
327.625	E7.2	327.625	No exponent subfield.
−.0003627+5	E11.7	−36.27	Integer subfield only a minus sign and a plus sign appears instead of E.
−.0003627E5	E11.7	−36.27	Integer subfield left of decimal contains minus sign only.
spaces	E4.1	0.	All subfields empty.

Output

The width **w** must be sufficient to contain digits, plus or minus signs, decimal point, E, the exponent, and spaces. Generally, the following values for **w** are sufficient:

w ≥ d + 6 or **w** ≥ d+ e + 4

for negative numbers or positive numbers under SP descriptor control.

w ≥ d + 5 or **w** ≥ d + e + 3

for positive numbers not under SP descriptor control.

Positive numbers need not reserve a space for the sign of the number unless an SP descriptor specification is in effect. If the field is not wide enough to contain the output value, asterisks are inserted throughout the field. If the field is longer than the output value, the quantity is right-justified with spaces on the left. If the value being converted is indefinite, an I is printed in the field; if it is infinite (out-of-range), an R is printed.

The Ew.d specification produces output in the following formats:

s.aE±ee

s.a±eee

s

Minus sign if the number is negative; omitted if the number is positive

a

One or more most significant digits of the value correctly rounded

ee

A 2-digit exponent

eee

A 3-digit exponent

The first form is used for values where the magnitude of the exponent is less than 100; the second form is used for values where the magnitude of the exponent is in the range 100 through 999. For values where the magnitude of the exponent exceeds 999, you must explicitly specify the exponent length (at least 4 digits) using the form Ew.dEe.

When the specification Ew.dEe is used, the exponent is preceded by E, and the number of digits used for the exponent field not counting the letter and sign is determined by **e**. If the value specified for **e** is too small for the value being output, the entire field width as specified by **w** is filled with asterisks.

If a real variable containing an integer value is output under the Ew.d specification, results are unpredictable since the internal formats of real and integer values differ. An integer value normally does not have an exponent and is printed, therefore, as a very small value or 0.0.

Following are some examples of Ew.d output.

Internal Value	Format Specification	Output Field
67.32	E9.3	Δ.673E+02
-67.32	E9.3	-.673E+02
67.32	E12.3	ΔΔΔΔ.673E+02
-67.32	E12.3	ΔΔΔ-.673E+02

F Descriptor

The F descriptor specifies conversion between an internal real or double precision number and an external floating-point number. This descriptor has the form:

Fw.d

Input

On input, the F specification is treated identically to the E specification. An exponent can appear in the input field.

Following are some examples of Fw.d input.

Input Field	Specification	Converted Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field.
.62543	F6.5	.62543	No integer subfield.
.62543	F6.2	.62543	Decimal point overrides d of Fw.d specification.
+144.15E−03	F11.2	.14415	Exponents are allowed in F input.
50000	F5.2	500.00	No fraction subfield; input number converted as 50,000x10**−2.
spaces	F5.2	0	Spaces in input field interpreted as 0.

Output

The F descriptor outputs a real number without a decimal exponent using the following format:

sn.n

n

Field of decimal digits

s

Minus sign if the number is negative, or omitted if the number is positive

If the field is too short, a field of asterisks is output. If the field is longer than required, the number is right-justified with spaces on the left. If the value being converted is indefinite, an I is printed in the field; if it is infinite (out-of-range), an R is printed.

Following are some examples of F output.

Internal Value	Format Specification	Output Field
+32.694	F6.3	32.694
+32.694	F10.3	ΔΔΔΔ32.694
-32.694	F6.3	******
.32694	F4.3	.327
32.694	F6.0	ΔΔΔ33.

G Descriptor

The G descriptor specifies conversion between an internal real or double precision number and an external floating-point number written either with or without an exponent. This descriptor has the forms:

Gw.d
Gw.dEe

Input

Input under control of the G specification is the same as for the E specification. The rules that apply to the E specification also apply to the G specification.

Output

Output under control of the G descriptor depends on the size of the floating-point number being edited. For values greater than or equal to .1 and less than 10**d in magnitude, the number is output under modified F format as shown below.

Size of Number N	F Format Conversion if Gw.d Is Used	F Format Conversion if Gw.d.Ee Is Used
0.1<=N<1	F(w−4).d+4 spaces	F(w−e−2).d+(w−e−2) spaces
1<= N<10	F(w−4).(d−1)+4 spaces	F(w−e−2.(d−1)+(w−e−2) spaces
⋮	⋮	⋮
10**(d−z)<=N<10**(d−1)	F(w−4).1+4 spaces	F(w−e−2).1+(w−e−2) spaces
10**(d−1)<= N<10**d	F(w−4).0+4 spaces	F(w−e−2).0+(w−e−2) spaces

For values outside this range, Gw.d output is identical to Ew.d, and Gw.dEe is identical to Ew.dEe. If the value being converted is indefinite, an I is printed in the field; if it is infinite (out-of-range), an R is printed.

If a number is output under the Gw.d specification without an exponent, four spaces are inserted to the right of the field (these spaces are reserved for the exponent field E±ee). Therefore, for output under G conversion, **w** must be greater than or equal to d + 6. The six extra spaces are required for sign, decimal point, and 4-space exponent field. If the Gw.dEe form is used for a number output without an exponent, then e + 2 spaces are inserted to the right of the field.

Following are some examples of G output.

Internal Value	G14.8 Output Field	Format Option
.001415926535	.14159265E−02	E conversion
.8979323846	.89793238	F conversion
2643383279.	.26433833E+10	E conversion
−693.9937510	−693.99375	F conversion

I Descriptor

The I descriptor specifies conversion from an internal number to integer. This descriptor has the forms:

Iw
Iw.m

Input

You can omit the plus sign for positive integers. When a sign appears, it must precede the first digit in the field. An Iw.m specification has no effect on input. The interpretation of spaces within a field depends on the BLANK= specifier on the OPEN statement (described later in this chapter). If this specifier is omitted, spaces are ignored. An all-blank field is always considered to be zero. Decimal points are not permitted. You cannot have any character other than a decimal digit, space, or the leading plus or minus sign in an integer field on input.

The following example shows Iw input. This example assumes that spaces are ignored. Spaces are ignored if the BLANK= specifier is omitted from OPEN statement or BLANK='NULL' is specified. (If BLANK='ZERO' were specified, spaces would be interpreted as zeros. In this case, J would contain −1500 and N would contain 104. The other values would not be affected.)

```
      OPEN (2, BLANK='NULL')
      READ (2, 10) I, J, K, L, M, N
10    FORMAT (I3, I7, I2, I3, I2, I4)
```

Input Record:

```
139|ΔΔ-15ΔΔ|18|ΔΔ7|ΔΔ|1Δ4
```

(Bars mark input fields.)

In Storage:

```
I: 139    L: 7
J: -15    M: 0
k: 18     N: 14
```

Output

If the integer is positive, the plus sign is suppressed unless an SP specification (described later in this chapter) is in effect. Leading zeros are suppressed.

If Iw.m is used and the output value occupies fewer than m positions, leading zeros are generated to fill up to m digits. If m=0, a zero value produces all spaces. If m=w, no spaces occur in the field when the value is positive, and the field is too short for any negative value. If the field is too short, asterisks occupy the field.

Following are some examples of I output. The first character of a printer output record is used for printer control and is not printed. More information on printer control appears under Printer Control Character in this chapter.

Example:

```
     PRINT 10, I, J, K
10   FORMAT (I9, I10, I5.3)
```

In Storage:

```
I: -3762
J: 4762937
K: 13
```

Printed Output:

ΔΔΔ-3762|ΔΔΔ4762937|ΔΔ013

(Bars mark output fields.)

The first character in a printed output line is always interpreted as a printer control character and does not appear in the output.

Example:

```
    WRITE (6, 100) N, M, J
100 FORMAT (I5, I6, I9)
```

In Storage:

```
N: 20
M: -731450
J: 205
```

Printed Output:

ΔΔ20|******|ΔΔΔΔΔΔ205

(Bars mark output fields.)

Asterisks are printed in the second output field because the specification, I6, is too small.

L Descriptor

The L descriptor is used to input or output logical data items. This descriptor has the form:

Lw

Input

If the first nonblank characters in the field are T or .T, the logical value true is stored in the corresponding *iolist* item, which should be of type logical. If the first nonblank characters are F or .F, the value false is stored. If the first nonblank characters are not T, .T, F, or .F, a diagnostic is printed. An all blank field has the value false.

Output

Variables output under the L specification should be of type logical. A value of true or false in memory is output as a right-justified T or F with blank fill on the left.

Example:

```
      LOGICAL I, J, K
      I=.TRUE.
      J=.FALSE.
      K=.TRUE.
      PRINT 5, I, J, K
   5  FORMAT (1X, 3L3)
```

These statements print the following:

```
ΔΔTΔΔFΔΔT
```

Control Data Extension

O Descriptor

The O descriptor is used to input or output items in octal format. This descriptor has the forms:

Ow
Ow.m

The two forms have the same meaning on input, but different meanings on output. The octal digits are the numbers 0 through 7.

Input

An input field corresponding to an integer, byte, real, logical, or boolean *iolist* item can contain a maximum of 22 digits. An input field corresponding to a complex or double precision *iolist* item can contain a maximum of 43 digits. Spaces are allowed and a plus or minus sign can precede the first octal digit. Spaces are interpreted as zeros and a field of all spaces is interpreted as zero. A decimal point is not allowed. The digits are stored right-justified within the *iolist* item, with zero fill on the left.

If an input field corresponding to an integer, byte, real, logical, or boolean *iolist* item contains 22 digits, the leftmost two bit positions of the input field must be zero. If an input field corresponding to a complex or double precision *iolist* item contains 43 digits, the leftmost bit position of the input field must be zero.

Example:

```
      BOOLEAN P, Q, R
      READ 10, P, Q, R
   10 FORMAT (O10, O12, O2)
```

Input Record:

```
3737373737|666Δ6644Δ444|-0
```

(Bars mark input fields as specified by the FORMAT statement.)

In Storage (octal representation):

```
P:  0000000000003737373737
Q:  0000000000666066440444
R:  0000000000000000000000
```

Control Data Extension *(Continued)*

Output

If **w** is less than or equal to 22 (for integer, byte, real, logical, or boolean *iolist* item) or 43 (for complex or double precision *iolist* item), the rightmost digits are output. The output field includes leading zeros. For example, if location P contains a value with the octal representation

```
0000000000003737373737
```

and the output statements are

```
    WRITE (6, 100) P
100 FORMAT (1X, O4)
```

then the digits 3737 are output.

If **w** is greater than 22 (for integer, byte, real, logical, or boolean list items), 22 digits are output right-justified with spaces on the left. For example, if the preceding format specification is changed to (1X, O24), output is as follows:

```
ΔΔ0000000000003737373737
```

If **w** is greater than 43 for complex or double precision *iolist* item, 43 digits are output right-justified with spaces on the left.

A negative number is output in two's complement internal form. For example, the statements

```
    I=-11
    PRINT 200, I
200 FORMAT (1X, O22)
```

produce the following output:

```
1777777777777777777765
```

The value of m causes leading zeros to be written as spaces. If m is specified, up to w−m leading zeros are written as spaces. If the number cannot be output in **w** octal digits, the rightmost w*3 bits are converted to octal and written.

R Descriptor

The R descriptor is used with any type *iolist* items. This descriptor transmits the rightmost characters of a word with no conversion. The R descriptor has the form:

Rw

On both input and output, the R specification is identical to the A specification unless **w** is less than 8 (for integer, byte, real, logical, or boolean *iolist* item) or 16 (for complex or double precision *iolist* item).

On input, if **w** is less than 8 (for integer, byte, real, logical, or boolean *iolist* item) or 16 (for complex or double precision *iolist* item), the rightmost **w** characters are read and stored right-justified with upper binary zero fill.

On output, if **w** is less than 8 (for integer, byte, real, logical or boolean *iolist* item) or 16 (for complex or double precision *iolist* item), the rightmost **w** characters of the output item are written to the output record.

Example of R input:

```
      BOOLEAN HOO, RAY
      READ (5, 600) HOO, RAY
  600 FORMAT (R8, R7)
```

Input Record:

```
RESULTSΔ|OF TEST
```

In Storage:

```
HOO:RESULTSΔ
RAY:0OFΔTEST
```

(Leftmost character is ASCII null, not ASCII zero)

Z Descriptor

The Z descriptor converts internal numbers to hexadecimal. This descriptor has the forms:

Zw
Zw.m

The form Zw.m is used for output only. Hexadecimal digits include the digits 0 through 9 and the letters A through F. A hexadecimal digit is represented by 4 bits.

Input

The input string can contain up to 16 hexadecimal digits (for an integer, byte, real, logical, or boolean *iolist* item) or 32 hexadecimal digits (for a complex or double precision *iolist* item). Embedded spaces are interpreted as zero and an all blank field is equivalent to zero. A plus or minus sign can precede the first digit. The string is stored right-justified within the *iolist* item, with zero fill on the left.

Example of Z input:

```
INTEGER R, S
READ (10, '(Z10, Z4)') R, S
```

Input Record:

```
A309FFFFCC4 D1
```

In Storage (hexadecimal representation):

```
R: 000000A309FFFFCC
S: 00000000000040D1
```

Output

If **w** is less than 16 (for integer, byte, real, logical, or boolean list items) or 32 (for complex or double precision *iolist* item), the rightmost w*4 bits are converted to hexadecimal and written. The output field includes leading zeros. For example, if location I contains

```
00000000000FB26C
```

then the output statement

```
WRITE(6, '(1X,Z3)') I
```

writes the digits 26C.

Control Data Extension *(Continued)*

If **w** is greater than 16 for integer, byte, real, logical, or boolean list items, the 16 hexadecimal digits are right-justified with spaces on the left.

If **w** is greater than 32 for complex or double precision *iolist* item, the 32 hexadecimal digits are right-justified with spaces on the left.

The value of m causes leading zeros to be written as spaces. If m is specified, up to w−m leading zeros are written as spaces. If the number cannot be written in **w** hexadecimal digits, a field of asterisks is written.

End of Control Data Extension

Nonrepeatable Edit Descriptors

The nonrepeatable edit descriptors control aspects of formatting, but do not cause data conversions and are not associated with items in the *iolist*. The nonrepeatable edit descriptors are BN, BZ, H, P, S, SS, SP, T, TL, TR, X, ', ", /, and :.

P Descriptor

The P descriptor has the form:

kP

k

Signed or unsigned integer constant called the scale factor.

The P descriptor changes the position of a decimal point of a real number when it is input or output. The descriptor kP causes subsequent format specifications to be scaled by 10**k. Scale factors can precede D, E, F, and G format specifications or appear independently:

kP	..d	kPEw.d	kPGw.d
kPEw.dEe	kPFw.d	kP	

A scale factor of zero is established when each format specification is first referenced; it holds for all D, E, F, and G edit descriptors until another scale factor is encountered.

Once a scale factor is specified, it holds for all D, E, F, and G descriptors in that FORMAT specification until another scale factor is encountered. To cancel the effect of a scale factor for subsequent D, E, F, and G descriptors, you must specify a zero scale factor (0P). For example, in the format specification

```
15  FORMAT (2P,E14.3,F10.2,G16.2,0P,4F13.2)
```

the 2P scale factor applies to the E14.3 descriptor and also to the F10.2 and G16.2 descriptors. The 0P scale factor restores normal scaling (10**0=1) for the subsequent specification 4F13.2.

Input

For D, E, F, and G editing, provided that the number in the input field does not have an exponent, the number is divided by 10**k and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is 314.1592 * 10**(-2)=3.141592. However, if the input number contains an exponent, the scale factor is ignored.

Output

For F editing, the number in the output field is the internal number multiplied by 10**k. In the output representation, the decimal point is fixed; the number is either left-justified or right-justified, depending on whether the scale factor is plus or minus. For example, the internal number -3.1415926536 can be represented on output under various scaled F specifications as follows:

Specification	Number Output
-1PF13.6	-.314159
F13.6	-3.141593
1PF13.6	-31.415927
3PF13.6	-3141.592654

For E and D editing, the effect of the scale factor kP is to shift the decimal point right k places and reduce the exponent by k. In addition, the scale factor controls the decimal normalization between the coefficient and the exponent as follows:

- If k is less than or equal to zero, there will be exactly -k leading zeros and d + k significant digits after the decimal point.
- If k is greater than zero, there will be exactly k significant digits to the left of the decimal point and d - k + 1 significant digits to the right of the decimal point.

For example, the number -3.1415926536 is represented on output under various Ew.d scaling as follows:

Specification	Number Output
-3P E20.4	-.0003E+04
-1P E20.4	-.0314E+02
E20.4	-.3142E+01
P E20.4	-3.1416E+00
3P E20.4	-314.16E-02

For G editing, the effect of the scale factor is cancelled unless the magnitude of the number to be output is outside the range that permits effective use of F conversion (namely, unless the number is less than 10**(-1) or equal to or greater than 10**d). In these cases, the scale factor has the same effect as described for Ew.d and Dw.d scaling. For example, the number -.00031415926536 is represented on output under the indicated Gw.d scaling as follows:

Specification	Number Output
-3PG2.6	-.000314E+00
-1G20.6	-.031416E-02
20.6	-.314159E-03
1G20.6	-3.141593E-04
3PG20.6	-314.1593E-06
5PG20.6	-31415.93E-08
7PG20.6	-3141593.E-10

The number -3.1415926536 would be output as -3.14159 under any of the preceding specifications, because that number is within the required range.

BN and BZ Descriptors

The BN and BZ descriptors can be used on input with the D, E, F, G, and I edit descriptors to specify the interpretation of spaces (other than leading spaces). In the BN or BZ descriptor is omitted, spaces in input fields are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier currently in effect for the input/output unit. BLANK='NULL' is the default for input. If a BN descriptor is encountered in a format specification, all space characters in succeeding numeric input fields are ignored; that is, the field is treated as if spaces had been removed, the remaining portion of the field right-justified, and the field padded with leading spaces. A field of all spaces has a value of zero.

If a BZ descriptor is encountered in a format specification, all space characters in succeeding numeric input fields are interpreted as zeros.

For example, assuming BLANK='NULL', if the statement

```
READ (6, '(I3, BZ, I3, BN, I3)') I, J, K
```

reads the input record

```
1ΔΔ2ΔΔ3ΔΔ
```

then I, J, and K have the following values:

```
I=1  J=200  K=3
```

S, SP, and SS Descriptors

The S, SP, and SS descriptors are used on output with the D, E, F, G, and I descriptors to control the printing of plus (+) characters. S, SP and SS have no effect on input.

Normally, FORTRAN does not precede positive numbers by a plus sign on output. If an SP descriptor is encountered in a format specification, all succeeding positive numeric fields contain the plus sign (w must be of sufficient length to include the sign). If an SS or S descriptor is encountered, the optional plus signs do not appear.

S, SP, and SS have no effect on plus signs preceding exponents because those signs are always provided. For example, the statements:

```
A=10.5
B=7.3
C=26.0
WRITE (2, '(1X, F6.2, SP, F6.2, SS, F6.2)') A, B, C
```

print the following:

```
Δ10.50Δ+7.30Δ26.00
```

H Descriptor

The H descriptor outputs strings of characters. This descriptor is not associated with a variable in the output iolist. The H descriptor has the form:

nHstring

n

Number of characters in the string, including spaces.

string

String of characters.

The H descriptor cannot be used on input.

Although using apostrophes to designate a character string precludes the need to count characters, the H descriptor may be more convenient if the string contains apostrophes. For example, the sequence

```
    A=1.5
    WRITE (2, 30) A
30  FORMAT (6H LMAX=, F5.2)
```

writes the following output:

```
ΔLMAX=Δ1.50
```

Replacing the H descriptor in the preceding example with ' LMAX=' would produce the same output.

Quote and Apostrophe Descriptors

Character strings delimited by a pair of apostrophe (') or quote (") symbols can be used as alternate forms of the H specification for output. The paired symbols delimit the string. The string must not be empty. You cannot use the apostrophe and quote descriptors on input.

NOTE

Because the apostrophe descriptor is ANSI standard, it is preferred over the quote descriptor.

The following example shows how to write a character string that is continued on a second line:

```
      WRITE (6, 20)
20    FORMAT (' RESULT OF CALCULATIONS IS ',
     *'AS FOLLOWS')
```

These statements produce the following output:

```
RESULT OF CALCULATIONS IS AS FOLLOWS
```

To represent an apostrophe or quote within a string delimited by the same symbol, use two consecutive occurrences of the symbol. Alternatively, if a quote or apostrophe appears within a string, the other symbol can be used as the delimiter. The following example shows two ways of writing a character string that contains an apostrophe:

```
    PRINT 1
    PRINT 2
1   FORMAT (" ABC'DE")
2   FORMAT (' DON''T')
```

These statements produce the following output:

```
ABC'DE
DON'T
```

X Descriptor

The X descriptor skips character positions in an input line or output line. X is not associated with an item in the *iolist* specifier. The X descriptor has the form:

nX

n

Number of character positions to be skipped from the current character position; n is a nonzero unsigned integer.

The specification nX indicates that transmission of the next character to or from a record is to occur at the position n characters forward from the current position. When an X specification causes control to pass over character positions on output, positions not previously filled during record generation are blank filled; those already filled are left unchanged.

Example:

```
A=-342.743
B=1.53190
J=22
WRITE (6, '(1X, F9.4, 4X, F7.5, 4X, I3)') A, B, J
```

Output:

```
Δ-342.7430ΔΔΔΔ1.53190ΔΔΔΔΔ22
```

Example:

```
READ (3, '(F5.2, 3X, F5.2, 6X, F5.2)') R, S, T
```

Input record:

```
14.62ΔΔ$13.78ΔCOSTΔ15.97
```

In R, S, and T:

```
R:  14.62  S:  13.78  T:  15.97
```

T, TL, and TR Descriptors

The T, TL, and TR descriptors provide for tabulation control. These descriptors have the forms:

Tn
TLn
TRn

n

Nonzero unsigned decimal integer

When a Tn descriptor is encountered in a format specification, input or output control skips right or left to character position n; the next edit descriptor is then processed.

When a TLn descriptor is encountered, control skips backward (left) n character positions. If n is greater than or equal to the current position, control skips to the first position.

When a TRn descriptor is encountered, control skips forward (right) n positions.

Example:

```
    READ 40, A, B, C
40  FORMAT (T2, F5.2, TR5, F6.1, TR3, F5.2)
```

Input record:

```
Δ684.73ΔΔΔΔΔ2436.2ΔΔΔΔ89.14
```

Stored in A, B, and C:

```
A: 684.7   B: 2436.0   C: 89.0
```

Example:

```
    WRITE (31, 10)
10  FORMAT (T20, 'LABELS')
```

The preceding statements position to character position 20 of the output record and write the characters LABELS.

With a T, TR, or TL specification, the order of a list need not be the same as that of the input or output record, and the same information can be read more than once. For example, if the statement:

```
READ (2, '(F5.2, TL5, F5.2)') A, B
```

reads the record:

```
76.05
```

then both A and B contain 76.05.

When a T, TR, or TL specification causes control to pass over character positions on output, positions not previously filled during record generation are set to spaces; those already filled are left unchanged.

It is possible to destroy a previously formed field. For example, the statements:

```
    PRINT 8
8   FORMAT (' DISASTERS', T5, 3H1 23)
```

print the following string:

```
DIS123ERS
```

Slash (End-of-Record) Descriptor

A slash in a format specification indicates the end of a record. One or more slashes can separate edit descriptors, a comma separator is allowed between slashes. Slashes can precede the first edit descriptor in a format specification, can follow the last edit descriptor, or can appear between edit descriptors.

On input, a slash specifies that further data comes from the next record. If a slash is the last descriptor, it causes an input record to be skipped.

On output, a slash causes subsequent data to be written to the next record. When a slash is the last descriptor, it causes a record of spaces to be written.

Example:

```
      DIMENSION B(3)
      READ (5, 100) IA, B
  100 FORMAT (I5/3E7.2)
```

These statements read two records; the first contains an integer number, and the second contains three real numbers.

Example:

```
      A=46.3272
      WRITE (3, 11) A
  11  FORMAT (1X, 'NEW VALUE', //1X, F7.3)
```

Printed output:

```
  NEW VALUE     (blank line)
  Δ46.327
```

Each line corresponds to a formatted record. The second record is blank and produces the line spacing shown.

Colon (:) Descriptor

A colon (:) in a format specification terminates format control if there are no more items in the *iolist* specifier. The colon has no effect if there are more items in the *iolist* specifier. This descriptor is useful in forms where non*iolist* item edit descriptors follow *iolist* item edit descriptors; when the *iolist* is exhausted, the subsequent edit descriptors are not processed. For example, the statements

```
      A=1.0
      B=2.2
      C=3.1
      D=5.7
      PRINT 10, A, B, C, D
  10  FORMAT (4(F4.1, :, ','))
```

print the following record:

```
  1.0, 2.2, 3.1, 5.7
```

In this example, format control terminates after the value of D is printed, and the last comma is not printed.

Unformatted Input/Output

Unformatted input/output statements do not use format specifications and do not convert data in any way on input or output. Instead, data is transferred as is between memory and the external device. (Since the data is in an internal form, it is generally not suitable for printing or terminal display.)

Each unformatted input/output statement transfers exactly one record.

When a null record is written to a file with the RECORD_TYPE=FIXED attribute, such as an unformatted direct access file, the record is filled with the padding character before being written. If the record is less than the MAXIMUM_RECORD_LENGTH attribute, then the unused portions are also filled with the padding character. File attributes are described under Default File Attributes in this chapter.

For sequential unformatted input/output, an error message is issued when an attempt is made to read more characters than the record contains.

The unformatted input/output statements described in this section are:

- Unformatted WRITE
- Unformatted READ

Unformatted WRITE

The unformatted WRITE statement is used to output records in their internal form. This statement has the form:

WRITE(*UNIT*=**u**, *IOSTAT=ios, ERR=sl) iolist*

The *UNIT=, IOSTAT=, and ERR=* specifiers are described earlier in this chapter under Input/Output Statement Specifiers. The *iolist* specifier is described under Input/Output Lists.

Information is transferred from the items in *iolist* to the specified output unit with no format conversion. One record is created by each unformatted WRITE statement. If *iolist* is omitted, the statement writes a null record on the output device. A null record is a record that contains no data.

Example:

```
DIMENSION A(260), B(4000)
      :
WRITE (10, ERR=16) A, B
```

The 4260 words of arrays A and B are written as one record in internal binary format on unit 10.

Unformatted READ

The unformatted READ statement transmits one record from the specified unit to the storage locations named in *iolist*. This statement has the form:

READ(*UNIT=* **u**, *IOSTAT=ios, ERR=sl, END=sl*) *iolist*

The *UNIT=, IOSTAT=, ERR=, and END=* specifiers are described earlier in this chapter under Input/Output Statement Specifiers. The *iolist* specifier is described under Input/Output Lists.

No format specification is used, and the transmitted data is not converted.

The number of words in the list cannot exceed the number of words in the record. If the number of locations specified in *iolist* is less than the number of words in the record, the excess data is ignored. If *iolist* is omitted, the unformatted READ skips one record.

You should specify the END= or IOSTAT= specifier to test for an end-of-file. (If neither is specified, and an end-of-file is encountered, the program terminates with a fatal error.) Do not attempt to read a unit after an END= or IOSTAT= specifier has returned an end-of-file condition for that unit. Records following an end-of-file can be read by using a CLOSE statement followed by an OPEN statement on the file or by using the EOF function.

Example:

```
READ (2, END=30, ERR=40) X, Y, Z
```

This statement reads numbers directly into X, Y, and Z with no conversions.

List Directed Input/Output

List directed input/output involves the conversion of records according to compiler-defined formatting rules (without an explicit format specification). Each record consists of a list of values in a less restricted format than is used for formatted input/output. This type of input/output is particularly convenient when the exact form of data is not important.

The list directed statements described in this section are:

- List directed READ
- List directed WRITE
- List directed PRINT
- List directed PUNCH

List Directed READ

The list directed READ statement transmits data from the specified unit or the unit INPUT (if u is omitted or UNIT= * is specified) to the storage locations named in *iolist*. The list directed READ statement has the forms:

READ(*UNIT*=**u**, *FMT=*, IOSTAT=ios, ERR=sl, END=sl) iolist*

READ *, iolist

The *UNIT=, FMT=, IOSTAT=, ERR=, and END=* specifiers are described earlier in this chapter under Input/Output Statement Specifiers. The *iolist* specifier is described under Input/Output Lists.

Unlike formatted input, in which *iolist* items are in fixed-length fields, input items for list directed input are free-form with separators.

A list directed READ following a list directed READ that terminated in the middle of a record starts with the next data record.

You should specify the END= or IOSTAT= specifier to test for an end-of-file. (If neither is specified, and an end-of-file is encountered, the program terminates with a fatal error.) Do not attempt to read a unit after an END= or IOSTAT= specifier has returned an end-of-file condition for that unit. Records following an end-of-file can be read by using a CLOSE statement followed by an OPEN statement on the file or by using the EOF function.

Input data consists of a string of values separated by one or more spaces, or by a comma or slash, either of which can be preceded or followed by any number of spaces. Also, a line boundary, such as the end of a terminal line or the end of a card, serves as a value separator; however, a separator adjacent to a line boundary does not indicate a null value.

Embedded spaces are not allowed in input values except in character values and between the components of complex numbers, as described below. The format of values in the input record is:

Byte numbers

Same format as for byte constants; values must not be out of range for byte constants.

Integers

Same format as for integer constants; 2- and 4-byte integers are allowed. Values must not be out of range for the associated integer constant.

Real numbers

Any valid FORTRAN format for real or double precision numbers. Sixteen-byte real values are allowed; values that are too large for an associated specification are not allowed. In addition, the decimal point can be omitted; it is assumed to be to the right of the number if no exponent is specified, or between the number and the exponent.

Complex numbers

Two real values, separated by a comma, and enclosed by parentheses. The parentheses are not considered to be a separator. The decimal point can be omitted from either of the real values. Each of the real values can be preceded or followed by spaces.

Character values

A string of characters (which can include spaces) enclosed by apostrophes. An apostrophe can be represented within a string by two successive apostrophes with no intervening characters. Character values can only be read into character arrays, array elements, variables and substrings. If the string length exceeds the length of the *iolist* item, the string is truncated. If the string is shorter than the *iolist* item, the string is left-justified and remaining character positions are blank filled.

Logical values

An optional period, followed by a T or F, followed by optional characters that do not include slashes, spaces, or commas. One-, 2-, 4-, and 8-byte values are allowed. (The logical constants .TRUE. and .FALSE. are valid.)

You can input a boolean constant only if the corresponding *iolist* item is of type boolean. Boolean constants include:

- Octal constants of the form O" ... ".
- Hexadecimal constants of the form Z" ... ".
- Hollerith constants containing one through eight characters and delimited by quotes. Constants of less than eight characters are left-justified with blank fill on the right.

In addition, real and integer values can be read into boolean variables.

An input item can be repeated by preceding the item by an integer repeat count and asterisk. For example, the input record

```
3*567.123
```

is equivalent to:

```
567.123,567.123,567.123
```

Spaces cannot immediately precede or follow the asterisk.

You can input a null in place of a constant when the current value of the corresponding *iolist* item is not to be changed. A null is indicated by a comma as the first character in the input string or by two commas separated by an arbitrary number of spaces. Nulls can be repeated by specifying an integer repeat count followed by an asterisk and any value separator. The next value begins immediately after a repeated null. You cannot use a null for either the real or imaginary part of a complex constant; however, a null can represent an entire complex constant.

When the value separator is a slash, the effect is the same as reading null values for the remaining input *iolist* items. The remainder of the current record is discarded.

Input values must correspond in type to variables in the *iolist* specifier. Integer input values must not be too large for an associated *iolist* item. For example, if *iolist* specifies a 2-byte integer variable, then the associated item in the input record must be within the range −32,768 to 32,767, which is the valid range for 2-byte integers.

The form of a real value can be the same as that of an integer value.

List Directed WRITE, PRINT, and PUNCH

The list directed output statements consist of a WRITE, a PRINT, and a PUNCH statement. These statements have the forms:

WRITE(*UNIT=* u, *FMT=* * , *IOSTAT=ios, ERR=sl*) *iolist*

PRINT *, iolist

PUNCH *, iolist

PRINT outputs data to the unit OUTPUT. PUNCH outputs to the unit PUNCH.

Data in the locations specified by *iolist* is converted from internal format to coded format and transferred to the designated unit.

List directed output is consistent with the input; however, comma separators, null values, slashes, repeated constants, and the apostrophes indicate character values are not produced. For real or double precision variables with absolute values in the range of 10**(-6) through 10**9, an F format conversion is used; otherwise, output has 1PE format. For real values, up to 13 digits are output. For double precision values, up to 27 digits are output. Trailing zeros in the fractional part and leading zeros in the exponent are suppressed.

List directed output statements always produce a space for printer control as the first character of the output record. The maximum length of an output line is the smaller of the PAGE_WIDTH and MAXIMUM_RECORD_LENGTH attribute values of the file. (These attributes can be set by the SET_FILE_ATTRIBUTE command, which is described under Changing File Attributes in this chapter.)

Logical values are output as T or F. Complex values are enclosed in parentheses with a comma separating the real and imaginary parts.

Boolean values are output in the form Z"n n ...", where n is a hexadecimal digit. Leading zeros are suppressed.

Example:

```
COMPLEX A
CHARACTER B*3
A=(7., - 1)
B='DOG'
C=123.45
PRINT *, A, B, C
```

These statements print the following record:

```
(7.,- 1.)DOG123.45
```

Control Data Extension

Namelist Input/Output

Namelist input/output permits formatted input and output of groups of variables and arrays by using an identifying group name instead of a format specification. The values are converted according to compiler-defined formatting rules. The name is established by the NAMELIST statement.

The namelist input/output statements described in this section are:

- NAMELIST statement
- Namelist READ
- Namelist WRITE
- Namelist PRINT
- Namelist PUNCH

NAMELIST Statement

The NAMELIST statement has the form:

NAMELIST / name / a, *..., a ... / name / a, ..., a*

name

Name to be given to the namelist group; must be unique within the program unit.

a

An 8-byte variable or array name. An array name must not specify an assumed-shape array.

The NAMELIST statement is a nonexecutable statement that appears in the declarative portion of the program following any specification statements. The namelist group name identifies the succeeding list of variable or array names.

You must declare a namelist group name in a NAMELIST statement before using the name in an input/output statement. The group name can be declared only once, and it cannot be used for any purpose other than a namelist name in the program unit. It can appear in READ, WRITE, PRINT, and PUNCH statements in place of the format specifier. When a namelist group name is used, the *iolist* must be omitted from the input/output statement.

A variable or array name can belong to one or more namelist groups.

Control Data Extension ***(Continued)***

Namelist READ

Namelist input is performed by the namelist READ statement. This statement has the form:

READ name

READ(*UNIT=* **u**, *FMT=* **name**, *IOSTAT=ios, ERR=sl, END=sl***)**

name

Name of the namelist group to be read.

The *UNIT=*, *FMT=*, *IOSTAT=*, *ERR=*, and *END=*specifiers are described earlier in this chapter under Input/Output Statement Specifiers.

When a READ statement references a namelist group name, input data in namelist format is read from the designated file. The specified group name must be found before an end-of-file is read. If the file is empty, an end-of-file condition results. This must be detected by an END= or IOSTAT= specifier. A subsequent read on the same file without an intervening positioning statement, CLOSE, OPEN, or EOF function check is not allowed.

Data read by a namelist READ statement must have the following namelist input group format:

$name item=value, ..., *item=value*, ***$END***

name

Namelist group name.

item=value

One of the following:

v=c

vc(i1:i2)=c

array*(s)***=r*c, ..., r*c**

carray(s)(i1:i2)=r*c, ..., r*c

where:

v

Variable name.

vc

Character variable name.

i1, i2

Integer constants representing the upper and lower bounds of a character substring.

Control Data Extension ***(Continued)***

c

Constant.

array

Array name.

carray

Character array name.

s

Array subscript in which each subscript expression is an integer constant; (s) is optional in the array(s) form, but is required in the carray(s)(i1:i2) form. The number of subscript expressions must be equal to the number of dimensions in the array.

r

Unsigned nonzero integer repetition factor; if omitted, * must also be omitted.

The form **r*c** is equivalent to **r** successive appearances of the constant **c**.

A & can be substituted for either of the $ characters in the group. Since the input operation terminates when the second $ or & is encountered, the characters END can be omitted.

In each record of a namelist group, position one is reserved for printer control and must be left blank. Data items following $name (or &name) are read until another $ (or &) is encountered.

Data read by a single namelist READ statement must contain only names listed in the referenced namelist group. All items in the namelist group, or any subset of the group, can be input. Values are unchanged for items not input. Variables need not be in the order in which they appear in the defining NAMELIST statement.

Spaces must not appear:

Between $ (or &) and the group name

Within array names and variable names

Spaces can be used freely elsewhere.

You can use more than one record as input data in a namelist group. The first position of each input record is ignored. All input records containing data should end with a constant followed by a comma; however, the last record can be terminated by a $ (or &) without the final comma. Each namelist group must begin in a new record. Constants can be preceded by a repetition factor followed by an asterisk.

Control Data Extension ***(Continued)***

Constants can be integer, byte, real, double precision, complex, logical, boolean, or character. Each constant must agree with the type of the corresponding input list item as follows:

- A logical, character, or complex constant must be of the same type as the corresponding input list item. A character constant is truncated from the right, or extended on the right with blanks, if necessary, to yield a constant of the same length as the variable, array element, or substring.

- An integer, byte, real, or double precision constant can be used for an integer, real, double precision, or boolean input list item. The constant is converted to the type of the list item. A boolean constant cannot be used for a non-boolean list item.

Logical constants have the following forms (lowercase forms are equivalent):

```
.TRUE.    .FALSE.
```

A character constant must have delimiting apostrophes. If a character constant occupies more than one record, each continuation of the constant must begin in column two; a complex constant has the form (c1,c2) where c1 and c2 are real constants. A character constant must extend to the end of a record preceding a continuation record. A boolean constant must be an octal constant, a hexadecimal constant, or a Hollerith constant delimited by quotes.

Spaces appearing within noncharacter constants are ignored. The BLANK= specifier in an OPEN statement has no effect on namelist input. A constant (other than a character constant) must not contain only spaces.

The following example illustrates a namelist input group:

Δ$AGRP	Group name
ΔXVAL=5.0,	Real number
ΔARR=5*(1.7,-2.4),	Five complex numbers
ΔCHAR='HI THERE',	Character string
Δ$END	Group terminator

Although the preceding example uses a separate input line for each variable or array definition, multiple definitions can be included on a single line.

Control Data Extension ***(Continued)***

Namelist WRITE, PRINT, and PUNCH

The namelist output statements consist of a WRITE statement, a PRINT statement, and a PUNCH statement. These statements have the following forms:

WRITE(*UNIT*=**u**, *FMT*=**name**, *IOSTAT*=*ios*, *ERR*=*sl*)

PRINT name

PUNCH name

name

Name of the namelist group to be written.

The *UNIT*=, *FMT*=, *IOSTAT*=, *ERR*=, and *END*=specifiers are described earlier in this chapter under Input/Output Statement Specifiers.

All variables and arrays and their values in the list associated with the namelist group name are output on the file associated with unit u, OUTPUT, or PUNCH. They are output in the order of specification in the NAMELIST statement. Output consists of at least three records. The first record is a $ in position 2 followed by the group name; the last record is a $ in position 2 followed by the characters END. Each group begins with triple spacing (a hyphen [-] inserted in the printer control position of each record).

No data appears in position 1 of any record of a namelist group. The maximum length of any output line is the smaller of the PAGE_WIDTH and MAXIMUM_RECORD_LENGTH attribute values of the file. (These attributes are described under Default File Attributes in this chapter.) Logical constants appear as T or F. Elements of an array are output in the order in which they were stored.

Character constants are written with delimiting apostrophes. Boolean constants are written in the form Z"n ... n", where n is a hexadecimal digit; leading zeros are suppressed.

Records output by a namelist WRITE statement can be read later in the same program by a namelist READ statement specifying the same group name.

Control Data Extension *(Continued)*

Following is an example of namelist input and output:

```
NAMELIST / INVAL / QUANT, COST
NAMELIST / OUTVAL / TOTAL, QUANT, COST
READ(*, INVAL)
TOTAL=QUANT * COST * 1.3
PRINT OUTVAL
```

Input Record:

```
'Δ$invalΔquant=1.5,Δcost=3.02Δ$'
```

Printed Output:

```
$OUTVAL
TOTAL=   .58889999999999E+01,
QUANT=.15E+01,
COST=.302E+01,
$END
```

The statement sequence defines two namelist groups, reads the first group, performs a simple calculation, and prints the second group.

Arrays in Namelist

Values can be read into an array by specifying, in the namelist input record, an array element followed by the values to be read, as follows:

array-element=constant, ..., *constant*

When data is input in this form, the constants are stored consecutively beginning with the location given by the array element. The number of constants can be less than or equal to, but must not exceed, the remaining number of elements in the array. For example:

```
INTEGER BAT(5)
NAMELIST / HAT / BAT, DOT
READ(*, HAT)
```

If the preceding statements read the following input record:

```
Δ$HAT BAT=2, 3, 3 * 4, DOT=1.05 $END
```

the value of DOT becomes 1.05 and the array BAT is as follows:

```
BAT(1) 2
BAT(2) 3
BAT(3) 4
BAT(4) 4
BAT(5) 4
```

End of Control Data Extension

Control Data Extension

Buffer Input/Output

NOTE

This feature is included for compatibility with other versions of FORTRAN, and its use is not recommended in new programs.

Buffer input/output transmits data between memory and an external storage device without conversion. Buffer input/output differs from unformatted reading and writing in that READ and WRITE statements are associated with an *iolist* specification. Buffer statements are not associated with a list; data is transmitted to or from a block of storage. Also, unlike unformatted READ operations, an attempt to buffer in more data than the record contains is allowed.

The ENDFILE, REWIND, and BACKSPACE statements are valid for files processed by buffer statements. However, a file processed by buffer statements cannot be processed in the same program by direct access input/output statements or by the mass storage subroutines.

Each buffer statement defines the location of the first and last words of the block of memory to or from which data is to be transmitted. The address of the last word must be greater than or equal to the address of the first word. The relative locations of the first and last word are defined only if they are the same variable or are in the same array, common block, or equivalence class. If the first and last words do not satisfy one of these relationships, their relative position is undefined and results are unpredictable.

If the first word and the last word are in the same common block but not in the same array or equivalence class, the operation will be successful but optimization might be degraded.

After a buffer statement has begun executing, the status of the buffer operation must be checked by a reference to the UNIT function. This reference must occur before referencing the same file or any of the contents of the block of memory to or from which data is transferred.

This status check ensures that the data has actually been transferred and the buffer parameters for the file have been restored. A second input/output operation must not occur on the same file without an intervening reference to UNIT.

Control Data Extension *(Continued)*

BUFFER IN

The BUFFER IN statement has the form:

BUFFER IN(u, p) (a, b)

u

Unit identifier. See the description of the unit identifier under Input/Output Statement Specifiers.

p

This parameter is provided for compatibility with previous versions of FORTRAN; it must be present and type integer, but it is disregarded.

a

First variable or array element of the block of memory to which data is to be transmitted; cannot be type character. Integer values must be 8- byte integers.

b

Last variable or array element of the block of memory to which data is to be transmitted; cannot be type character. Integer values must be 8- byte integers.

BUFFER IN transfers one record from the file indicated by u to the block of memory beginning at a and ending at b. If the record is shorter than the block of memory, excess locations are not changed. If the record is longer than the block of memory, excess words in the record are ignored, except when the record type is fixed (RECORD_TYPE=FIXED on the SET_FILE_ATTRIBUTE command), in which case an error occurs.

The UNIT function can test for an end-of-file condition after a BUFFER IN operation. After UNIT has been referenced, the number of words transferred to memory can be obtained by a call to the function LENGTH. If records do not terminate on a word boundary (in a file not written by BUFFER OUT), the exact length of the record is returned by LENGTHX in terms of words and excess bits or by LENGTHB in terms of bytes.

Example:

```
DIMENSION CALC(51)
BUFFER IN(1,P) (CALC(1), CALC(51))
IF(UNIT(1) .GE. 0) GO TO 20
```

Data is transferred from logical unit 1 into storage beginning at the first word of the array, CALC(1), and extending through CALC(51). An error or endfile condition transfers control to statement 20.

Control Data Extension *(Continued)*

BUFFER OUT

The BUFFER OUT statement has the form:

BUFFER OUT(u, p) (a, b)

u

Unit identifier. See the description of the unit identifier under Input/Output Statement Specifiers.

p

This parameter is provided for compatibility with previous versions of FORTRAN; it must be present and type integer, but it is disregarded.

a

First variable or array element of the block of memory from which data is to be transmitted; cannot be type character. Integer values must be 8- byte integers.

b

Last variable or array element of the block of memory from which data is to be transmitted; cannot be type character. Integer values must be 8- byte integers.

BUFFER OUT writes one record by transferring the contents of the block of memory beginning at a and ending at b to the file indicated by u. The length of the record is given by:

```
lwa - fwa + 1
```

where lwa is the last word address of the block of memory and fwa is the first word address. For fixed-length records (RECORD_TYPE=FIXED attribute), the record length is the length (number of bytes) specified by the MAXIMUM_RECORD_LENGTH parameter on the SET_FILE_ATTRIBUTE command. The specified length must not be greater than (lwa - fwa + 1) * 8.

Following a BUFFER OUT, the UNIT function must be referenced before another reference is made to the file or to the contents of the block of memory.

End of Control Data Extension

Control Data Extension

Mass Storage Input/Output

Mass storage input/output (MSIO) subroutines allow you to create, access, and modify files on a random basis without regard for their physical positioning. These files are called random files. Each record in a random file can be read or written at random without logically affecting the remaining file contents. You determine the length and content of each record. A random file can reside on any mass storage device.

A random file processed by mass storage subroutines should not be processed by any other form of input/output.

All integer arguments to the mass storage routines must be 8-byte integers.

Random Files

Random files offer the same advantages as direct access files. In a random file, as in a direct access file, any record can be read, written, or rewritten directly, because the file resides on a random access mass storage device that can be positioned to any record of a file.

NOTE

Direct access READ and WRITE, mass storage I/O routines, and the keyed-file interface subprograms all offer the advantages of random access. While the direct access capability is ANSI standard, it allows only fixed- length records. Keyed-file and mass storage input/output allow both fixed- and variable-length records.

To permit random access, each record in a random file is uniquely and permanently identified by a record key. A key is a value that you specify as a parameter on the call to read or write a record. When a record is first written, the key in the call becomes the permanent identifier for that record. The record can be retrieved later by a read call that specifies the same key, and it can be updated by a write call with the same key.

When a random file is in active use, the record key information is kept in an array. You are responsible for allocating the array space by a COMMON, DIMENSION, or type statement, but you must not change the array contents. The array should be noncharacter. Integer arrays must be 8-byte integer arrays. The array becomes the directory, or index, to the file contents. In addition to the record key information, it contains the word address and length of each record in the file. The index is the logical link that allows the mass storage subroutines to associate a user call key with the location of the required record.

The index is maintained automatically by the mass storage subroutines. You must not change the contents of the array containing the index in any manner, because to do so might result in destruction of the file contents. Before using the array as a subindex, you must set the array elements to zero and, if an existing file is being reopened and manipulated, read the subindex into the array.

In response to a call to the file, the assigned index array is automatically cleared. If an existing file is being reopened, the mass storage subroutines locate the master index in mass storage and read it into this array. Subsequent file manipulations make new index entries or update current entries. When the file is closed, the master index is written from the array to mass storage. When the file is reopened by the same job or another job, the index is again read into the index array, so that file manipulation can continue.

Index Key Types

There are two types of index keys: name and number. A name key can be specified as any nonzero boolean or 8-byte integer expression. A number key must be expressed as an 8-byte integer expression with a value greater than 0 and less than or equal to the length of the index in words, minus 1 word. You select the type of key by the *t* parameter of the OPENMS call. The key type selection is permanent. There is no way to change the key type, because of differences in the internal index structure. Do not reopen an existing file with an incorrect index type parameter. In addition, do not mix key types within an index. Mixed key types might result in destruction of a file.

For Better Performance

Where possible, the number key type is preferable to the name key type, because the program executes faster and require less storage. Faster execution occurs because it is not necessary for the I/O routines to search the index for a matching key entry (as is necessary when a name key is used). Space is saved because of the smaller index array length requirement.

Master Index

The master index type for a given file is selected by the *t* parameter in the OPENMS call when the index is created. The type cannot be changed after the file is created.

Subindex

The subindex type can be specified for each subindex. A different subindex name or number type can be specified by including the *t* parameter in the STINDX call. If *t* is omitted, the index type remains the same as the current index. Subsequent calls which omit the t parameter do not change the most recent explicit type specification; the type remains in effect until changed by another STINDX call.

STINDX cannot change the type of an index that already exists on a file. You must ensure that the *t* parameter in a call to an existing index agrees with the type of the index in the file. You must specify the correct subindex type.

Multilevel File Indexing

When a file is opened by an OPENMS call, the array selected as the index is cleared. If the call references an existing file, the file index is located and read into the array. This initializes the master index.

You can create additional indexes (subindexes) by allocating additional index arrays, preparing the arrays for use as described below, and calling the STINDX subroutine to indicate to the internal routines the location, length, and type of the subindex array. This process can be repeated as many times as required. (Each active subindex requires a separate index array area.) The mass storage routines use the subindex just as they use the master index; no distinction is made.

A separate array must be declared for each subindex that will be in active use. Inactive subindexes can be stored in the random file as additional data records.

The subindex is read from and written to the file by the standard READMS and WRITMS calls, in the same manner as any other data record. Although the master index array is cleared by OPENMS when the file is opened, STINDX does not clear the subindex array; therefore, you should clear the array to zeros before calling STINDX.

If an existing file is being manipulated and the subindex already exists on the file, you must read the subindex from the file into the subindex array by a call to READMS before STINDX is called. The STINDX call then directs the mass storage routines to use this subindex as the current index. The first WRITMS to an existing file using a subindex must be preceded by a call to STINDX to inform the mass storage routine where to place the index control word entry before the write takes place.

If you wish to retain the subindex, you must write it to the file after the current index designation has been changed back to the master index, or to a higher level subindex, by a call to STINDX.

OPENMS

The OPENMS call opens a mass storage file and informs the system that it is a random file. This call has the form:

CALL OPENMS(u, ix, len, t, *fn*)

u

Unit identifier.

ix

Array to contain the master index. It cannot be an assumed-shape array.

len

Integer expression specifying the length (in words) of the master index. For a number index:

len ≥ (number of entries) + 1

For a name index:

len ≥ 2 * (number of entries) + 1

t

Integer expression specifying index type; can have one of the following values:

0 File has a number master index.

1 File has a name master index.

fn

NOS/VE file reference specifying the file to opened. The file is connected to unit u.

If a file having the name derived from u does not already exist, a new file is created.

The array **ix** specified in the call is automatically cleared to zeros. If an existing file is being reopened, the master index is read from mass storage into the index array.

Example:

```
DIMENSION A(11)
CALL OPENMS(5, A, 11, 0)
```

These statements open a random file named TAPE5 using an 11-word (10-entry) master index of the number type. If the file already exists, the master index is read into memory starting at location A.

WRITMS

The WRITMS call transmits data from memory to a random file. This call has the form:

CALL WRITMS(u, arr, n, k, *r*, *s*)

u

Unit identifier.

arr

Name of array from which data is written. It cannot be an assumed-shape array.

n

Integer expression specifying the number of consecutive words to be written.

k

Record key. For number index, k can be any arithmetic expression whose value is:

$1 \le k \le len-1$

where *len* is the length of the master index. (See OPENMS call.)

For name index, k can be any character, boolean or integer expression. If k is an integer expression, the value BOOL(k) is used.

r

Rewrite specifier; integer expression having one of the following values:

−1	Rewrite in place if new record length does not exceed old record length; otherwise, write at end-of-data.
0	No rewrite; write at end-of-data.
1	Unconditional rewrite in place. The new record length must not exceed the old record length.

If *r* is omitted, 0 is used.

s

Subindex flag specifier; integer expression having one of the following values:

0	Do not subindex marker flag in index control word.
1	Write subindex marker flag in index control word for this record.

If *s* is omitted, 0 is used.

The end-of-data (for r=−1 and r=0) is defined to be immediately after the end of the data record which is closest to end-of-information. A random file record can be written in place, in which case it replaces an existing record, or it can be written at end-of-data.

Control Data Extension *(Continued)*

The *r* parameter can be omitted if the *s* parameter is also omitted. The *s* parameter marks a subindex record so that it can be distinguished from a data record.

Example:

```
CALL WRITMS(3, DATA, 25, 6, 1)
```

This statement unconditionally rewrites in place a 25-word record, having an index number key of 6, from array DATA to file TAPE3. The default value is taken for the *s* parameter.

READMS

The READMS call transmits data from the specified random file to memory. This statement has the form:

CALL READMS(u, arr, n, k)

u

Unit identifier.

arr

Name of array into which record is to be read. It cannot be an assumed-shape array.

n

Integer expression specifying the number of words to be read. If n is less than the record length, n words are read and no message is issued.

k

Record key specifying the record to be read. Specified key must match one of the keys defined in a WRITMS call.

Example:

```
CALL READMS(3, MORDAT, 25, 2)
```

This statement reads the first 25 words of record 2 from unit 3 (TAPE3) into memory starting at the first word of the array MORDAT.

CLOSMS

The CLOSMS call writes the master index from memory to the file and closes the file. This call has the form:

CALL CLOSMS(u)

u

Unit identifier

If CLOSMS is not explicitly called, an automatic CLOSMS occurs upon program termination for each random file opened by the program. However, CLOSMS allows you to close a file before the end of a run in order to associate a different file with the same unit.

Since new data records can overwrite the old master index, a file that has had new data records added is invalid unless the file is closed.

Example:

```
CALL CLOSMS(L "AFILE")
```

This statement closes the file AFILE.

STINDX

STINDX selects an array, other than the one specified in the OPENMS call, to be used as the current index to the file. This call has the form:

CALL STINDX(u, ixarr, len, t)

u

Unit identifier.

ixarr

Noncharacter array to contain the subindex. It cannot be an assumed-shape array.

len

Integer expression specifying the length, in words, of the subindex. For a number index:

len > number of entries + 1

For a name index:

len > 2 * (number of entries) + 1

t

Integer expression specifying the type of index. Must have one of the following values:

0 Number index.

1 Name index.

This argument can be omitted, in which case the index has the same type as the previously-specified index.

The call permits a file to be manipulated with more than one index. For example, when you wish to use a subindex instead of the master index, STINDX is called to select the subindex as the current index. The STINDX call does not cause the subindex to be read or written; that task must be performed by explicit READMS or WRITMS calls. STINDX merely updates the internal description of the current index to the file.

Example:

```
DIMENSION SUBIX(10)
CALL STINDX(3, SUBIX, 10, 0)
```

These statements select a new index, SUBIX, for file TAPE3 with an index length of 10 (up to 9 entries). The records referenced via this subindex use number keys.

Example:

```
DIMENSION MASTER(5)
CALL STINDX(2, MASTER, 5)
```

These statements select a new index, MASTER, for file TAPE2 with an index length of 5 and index type unchanged from the last index used.

Control Data Extension *(Continued)*

The following program creates a random file with a name index. The key names are RECORD1, RECORD2, RECORD3, and RECORD4:

```
      PROGRAM MS3
      DIMENSION INDEX(9), A(15, 4)
      CHARACTER*7 REC1, REC2
      DATA REC1/'RECORD1'/, REC2/'RECORD2'/
C
      CALL OPENMS (7, INDEX, 9, 1)
        :  (Generate data in array A)
C     WRITE 4 RECORDS TO FILE TAPE7.  THE KEY
C     NAMES ARE RECORD1, RECORD2, RECORD3, AND RECORD4.
      CALL WRITMS (7, A(1,1), 15, REC1)
      CALL WRITMS (7, A(1,2), 15, REC2)
      CALL WRITMS (7, A(1,3), 15, 'RECORD3')
      CALL WRITMS (7, A(1,4), 15, 'RECORD4')
C
C     CLOSE THE FILE
      CALL CLOSMS (7)
      END
```

End of Control Data Extension

Direct Access Files

Direct access files differ from conventional sequential files. In a sequential file:

- Records are stored in the order in which they are written and can normally be read back only in the same order.
- A record can be retrieved from a sequential file only by sequentially reading records until the desired record is read.
- Sequential operations can be slow and inconvenient in applications where the logical order of writing and of retrieving records differs.
- A sequential read requires a continuous awareness of the current file position and the position of the required record.

To circumvent these limitations, you can use direct access files. To read and write to a direct access file, use the formatted or unformatted input/output statements. Direct access files are created by a specifier on the OPEN statement as described under Creating A Direct Access File.

In a direct access file, any record can be read, written, or rewritten directly, without concern for the position or structure of the file. This is possible because the file resides on a random access mass storage device that can be positioned to any portion of a file without the need for a sequential search. Thus, the concept of file position does not apply to a direct access file. The notion of rewinding a direct access file is, for instance, without meaning.

You can use a direct access file for formatted or unformatted input/output. However, you cannot use a direct access file for list directed or namelist input/output.

You cannot use internal files for direct access input/output.

Records in a direct access file are identified by a record number. The record number is a nonzero positive decimal integer that is assigned when the record is written. Once a record is written with a record number, the record can always be accessed by referencing the same number. The order of records on a direct access file is the order of their record numbers. Records can be written, rewritten, or read by specifying the record number in a READ or WRITE statement. Records can be read or written in any order; they need not be referenced in the order of their record numbers. The number of the record is specified with the REC= specifier in a READ or WRITE statement.

Direct access input/output statements process byte-addressable files with F type records. F is the only record type permitted for direct access input/output.

An attempt to exceed the record length of a direct access file results in an error message.

Unlike other methods of random access, direct access files do not use a record key index. The disk address of the records in a direct access file are based on the product of the record number and record length. Thus, an arbitrarily large record number could cause a file to be excessively large. In general, the record ordinal should be used as the record number (first record numbered 1, second record numbered 2, and so forth.)

If the length of the *iolist* in a direct access formatted WRITE statement is less than the record length of the direct access file, the unused portion of the record is space filled. (You can select a different padding character through system commands.) A direct access WRITE statement must not attempt to write a record longer than the record length.

Creating a Direct Access File

To create a direct access file, you must specify an OPEN statement with the ACCESS='DIRECT' option. You must also specify the record length with the RECL= specifier in the OPEN statement. For example, the following statement opens an unformatted file named DAFL for direct access:

```
OPEN (2, FILE='DAFL', ACCESS='DIRECT', RECL=120)
```

The file is associated with unit 2 and has a record length of 120 words.

Direct Access File Examples

The following example writes variables A, B, and C to record number 6, and variables I, J, and X to record number 1 of the direct access file associated with unit 2:

```
WRITE (2, '(3E10.4)', REC=6) A, B, C
WRITE (2, '(2I4, G20.10)', REC=1) I, J, X
```

The following example reads records 10, 8, 6, 4, and 2 from direct access file DARG:

```
    OPEN (UNIT=2, FILE='DARG', ACCESS='DIRECT', FORM='FORMATTED', RECL=72)
    DO 14 I=10, 2, -2
    READ (2, 99, REC=I, ERR=20) (A(J), J=1, 6)
99  FORMAT (6E12.6)
      :
14  CONTINUE
```

Records 10, 8, 6, 4, and 2 are read from the direct access file DARG.

Direct Access Record Length Calculation

The record length for a formatted direct access file is specified in characters. The record length for an unformatted direct access file is specified in words. If the *iolist* for an unformatted WRITE contains character data, you still specify the record length to be written in words. You can determine the record length by the following rules:

1. Count each noncharacter item as 8 characters except for double precision and complex items, which count as 16 characters.

2. Calculate the total number of characters in all the character items.

3. Add the lengths calculated in steps 1 and 2 to determine the record length in characters; add 7, divide the result by 8, and truncate the fractional part to determine the number of words in the record.

Example:

```
CHARACTER A*7, B*9, C*10, D*20, E*15, F*12
INTEGER IA, IB, IC, ID(5)
OPEN (5, ACCESS='DIRECT', FORM='UNFORMATTED', RECL=17)
WRITE (5, REC=1) A, B, IA, C, IB, E, D, ID, F
```

The length of the output record is calculated as follows:

- Length of noncharacter items:
  length of IA=8 characters
  length of IB=8 characters
  length of ID=40 characters
  length of IC=8 characters
  Total length of noncharacter items=64 characters.

- Length of character items:
  length of A=7 characters
  length of B=9 characters
  length of C=10 characters
  length of D=20 characters
  length of E=15 characters
  length of F=12 characters
  Total length of character items is 73 characters.

- Record length in words=(56 + 73 + 7)/8=18 words.

Internal Input/Output

Internal input/output is performed on internal files. Internal files provide a means of reformatting and transferring data from one area of memory to another. Input and output on internal files are performed by formatted READ and WRITE statements and the ENCODE and DECODE statements. However, no input/output devices are involved.

Internal files allow data to be reformatted without the necessity of physically writing it and rereading it under a different format specification. Internal files also allow numeric conversion to or from character data type.

The two types of internal files are standard internal files and extended internal files. Standard internal files are preferred over extended internal files because they conform to the ANSI standard and are not dependent on the length of a computer word.

Standard Internal Files

A standard internal file can be any character variable, substring, or array (except an assumed-shape array). The variable, array, or substring is considered an internal file when it is referenced as an internal file in a READ or WRITE statement. If the file is a variable or substring, it consists of a single record whose length is the length of the variable or substring. If the file is an array, each array element constitutes a single record. For example, the declarative statement:

```
CHARACTER*20 A(100)
```

defines a character array containing 100 20-character elements. This array can be referenced as an internal file named A, containing 100 records of 20 characters each.

Records of an internal file are defined by storing data into the records, either with an output statement or an assignment statement.

The following restrictions apply to internal files:

- You cannot declare internal files in PROGRAM or OPEN statements.
- You can use only formatted input/output. Unformatted, list directed, namelist, mass storage, and buffer input/output cannot be user for internal files.
- You cannot use the file positioning or file status statements with internal files.

The standard internal input/output statements described in this section are

- Internal WRITE
- Internal READ

Internal WRITE

Data is written to standard internal files using a formatted WRITE statement in which the internal unit specifier u is a character variable, array, array element, or substring name. The WRITE statement transmits data from the variables specified in *iolist* to consecutive locations starting with the leftmost character of the location specified by u; data is converted from internal to character format according to the format specification. The number of characters transmitted is determined by the record length.

The following examples shows internal files used for output:

```
INTEGER A, B, C, D
CHARACTER*4 AR(4)
      ⋮
A=123
B=-27
C=104
D=1234
WRITE(AR, '(I4)') A, B, C, D
```

In array AR after the WRITE statement is executed:

```
AR(1):  'Δ123'   AR(2):  'Δ-27'   AR(3):  'Δ104'   AR(4):  '1234'
```

The WRITE statement defines an internal file, AR, and writes four records to the file. (The format specification I4 applies to each variable.

Example:

```
CHARACTER*8 FMT
DATA FMT /'(   F8.2)'/
      ⋮
WRITE(FMT(2:3), '(I2)') N
READ(1, FMT) (A(I), I=1, N)
```

This example builds a format specification at runtime. An N-element array is read according to format F8.2. The repeat count is specified at runtime. The internal WRITE statement writes the variable N to the second and third positions of the string (ΔΔF8.2). When the READ statement is executed, the format specification has the appropriate repeat count.

Internal READ

Data is read from a standard internal file using a formatted READ statement in which the internal unit identifier is a character variable, array, array element, or substring. Data is transferred from consecutive locations starting at the first character position of u, converted under format specification, and stored in the variables specified in *iolist.*

Example:

```
CHARACTER*3 ZT(4), A, B, C
      :
READ(ZT, '(A3)') A, B, C
```

Contents of array ZT:

```
CAT  DOG  RUN  CAR
```

Contents of A, B, and C after the READ statement:

```
A:  CAT
B:  DOG
C:  RUN
```

The READ statement reads three records from internal file ZT and stores data into variables A, B, and C. (The format specification A3 applies to all three variables.)

Example:

```
CHARACTER CN*12
      :
READ(CN, '(4I3)') I, J, K, L
```

Contents of CN:

```
2ΔΔΔ56Δ4ΔΔΔ8
```

Read Into I, J, K, and L (internal integer format):

```
I:  2
J:  56
K:  4
L:  8
```

The READ statement reads the single record of internal file CN, converts the data to internal integer format, and stores the converted data into variables I, J, K, and L.

Control Data Extension

Extended Internal Files

An extended internal file can be any noncharacter variable, array (except an assumed-shape array), or array element. A record of an extended internal file is defined by writing the record. The record length is measured in characters. Since one word contains eight characters, the record length of an extended internal file is given by:

8 * a

where a is the number of words in the record.

The extended internal input/output statements described in this section are:

- ENCODE
- DECODE

NOTE

This feature is included for compatibility with other versions of FORTRAN.

ENCODE

The ENCODE statement is the extended internal file output statement. This statement has the form:

ENCODE(c, fn, u) iolist

c

An unsigned, 8-byte, integer constant or variable having a value greater than zero; c specifies the number of characters to be transferred per record. The record length is calculated from c.

fn

The label of a FORMAT statement or a character expression whose value is a format specification; fn must not specify namelist or list directed formatting.

u

Identifies the extended internal file in which the record is to be encoded; u must be a noncharacter variable, array element, or array name.

iolist

A list of noncharacter variables, arrays, or array elements to be transmitted to the extended internal file identified by u.

ENCODE is similar to an internal file formatted WRITE. Values are transferred to the receiving storage area from the variables specified in **iolist** under the specified format. Integer values must be 8- byte integers. The first record starts with the leftmost character of the location specified by **u**. The length in characters of each record is given by:

```
INT((c + 7) / 8) * 8
```

where INT(a) is the largest integer less than or equal to a. If c is less than the record length, the remainder of the word is blank filled.

The internal file must be large enough to contain the total number of characters transmitted by the ENCODE statement. For example, if 56 characters are generated by the ENCODE statement, the array starting at location u must be at least 56 characters (seven words) in length. If A is the receiving array the declaration BOOLEAN A(7) is sufficient. If 27 characters are generated, the declaration BOOLEAN A(4) is sufficient.

The list and the format specification must not transmit more than the number of characters specified per record. If the number of characters transmitted is less than the record length, remaining characters in the record are blank filled.

You should not encode or decode an area in memory upon itself; the results are unpredictable.

DECODE

The DECODE statement is the extended internal file input statement. This statement has the form:

DECODE(c, fn, u) iolist

c

An unsigned, 8-byte, integer constant or variable having a value greater than zero; c specifies the number of characters to be transferred per record. The record length is calculated from c.

fn

The label of a FORMAT statement, or a character expression whose value is a format specification; fn must not specify namelist or list directed formatting.

u

Identifies the extended internal file in which the record is to be encoded; u is a noncharacter variable, array element, or array name.

iolist

A list of noncharacter variables, arrays, or array elements to be transmitted to the extended internal file identified by u.

iolist

A list of noncharacter variables, arrays, or array elements to receive data from the extended internal file identified by u.

DECODE is similar to an internal file formatted READ. Values are transferred to the variables specified in **iolist** under the specified format from the storage area. For integer type data, only 8-byte integers are allowed.

DECODE cannot process an illegal character for a given conversion specification.

DECODE can pack the partial contents of two words into one. Assume that two variables, LOC1 and LOC2, contain the following character values:

LOC1 SSSSΔΔΔΔ

LOC2 ΔΔΔΔDDDD

Control Data Extension ***(Continued)***

The following statements store the string 'SSSSDDDD' into the variable NAME:

```
      BOOLEAN LOC1, LOC2, TEMP, NAME
        :
      DECODE(8, 1, LOC2) TEMP
  1   FORMAT(4X, A4)
      ENCODE(8, 2, NAME) LOC1, TEMP
  2   FORMAT(2A4)
```

The DECODE statement places the last four display code characters of LOC2 into the first four characters of TEMP. The ENCODE statement packs the first four characters of LOC1 and TEMP into NAME.

End of Control Data Extension

Control Data Extension

Segment Access Files

The segment access file capability provides an efficient means of sharing large, randomly accessed blocks of data among applications. A named common block is associated (mapped) to a segment access file. You access the file directly through the common block's variables and arrays using assignment statements. The variables and arrays in the common block represent locations within the file.

You associate a common block with a segment access file by first specifying the common block name in a C$ SEGFILE directive and then using the OPEN statement specifying the common block name as the UNIT specifier (C$ SEGFILE is described in chapter 11, C$ Directives). When using the OPEN statement to associate a file with a common block, the UNIT and FILE specifiers are both required.

The UNIT specifier must specify the name of the common block, including slashes. The FILE specifier must specify the name of the segment access file you wish to associate with the common block. The IOSTAT, ERR and STATUS specifiers are optional. All other specifiers for the OPEN statement are not valid with segment access files.

You cannot reference a segment access file until after the OPEN statement has opened and associated the file with a common block. Therefore, you should initialize values in segment access files after the OPEN statement.

Segment access files that are associated with a common block are closed using the CLOSE statement. The SIZE parameter is an optional specifier on the CLOSE statement that can specify the length of the segment access file in bytes.

If a segment access file or associated common block is specified on an INQUIRE statement, then only the IOSTAT, ERR, EXIST, OPENED, NAMED and NAME specifiers can be used. The NAME specifier returns the common block name (including slashes) on an inquire by file and returns the name of the file on an inquire by unit. The NAMED specifier is true on an inquire by file and is the same as the value returned for the OPENED specifier on an inquire by unit.

To create and open a segment access file, specify the common block name as the UNIT specifier and the segment access file name as the FILE specifier (the FILE specifier is required for segment access files). For example:

```
      CHARACTER CH*5000
      COMMON / CHBLK / CH
C$    SEGFILE(/ CHBLK /)
      OPEN(UNIT=/ CHBLK /,
     + FILE='SEG_FILE_1',
     + ERR=10, IOSTAT=IVAR)
      CH(1:1)='A'
```

This creates and opens file SEG_FILE_1 and associates the file with common block CHBLK. The assignment statement sets the first byte of the file to the value of the character A.

For Better Performance

Segment access input/output is faster than other types of input/output. Your program may automatically use segment access files on a file that is not shared and is generally used or attached with default values. See Fast I/O in this chapter for more information.

End of Control Data Extension

Input/Output-Related Statements and Routines

This section describes the statements used to position files, to return status information about files and input/output operations, and to perform other tasks related to input/output. These statements are not for use with files processed by the keyed-file interface subprograms.

File Status Statements

FORTRAN provides three statements that can establish, examine, or alter certain attributes of files used for input or output. These are the OPEN, INQUIRE, and CLOSE statements.

OPEN

You can use the OPEN statement to associate an existing file with a unit number, to create a new file and associate it with a unit number, or to change certain attributes of an existing file. The OPEN statement has the form:

OPEN(*UNIT=* **u**, *IOSTAT=ios, ERR=sl, FILE=fin, STATUS=stat, ACCESS=acc, FORM=fm, RECL=rl, BLANK=blnk, BUFL=bl***)**

u

Unit identifier specifying the unit to be opened. If FILE= is also specified, the unit becomes associated with that file. For a segment access file, the unit identifier specifies the named common block (including slashes) to be associated with the FILE specifier.

ios

Specifies an 8-byte integer variable or array element into which one of the following values is returned after the input/output operation is complete:

Value Returned	Status of Input/Output Operation
< 0	End-of-file encountered.
= 0	Operation completed normally.
> 0	An error occurred. The value is either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier and its condition number.

sl

Label of an executable statement to which control transfers if an error occurs during the open.

fin

Character expression whose value is a file reference (512 or fewer characters) specifying the file to be opened. Trailing blanks are removed. This file becomes associated with unit **u**. If a file path is not specified, the file is assumed to be in the working catalog. However, if the file reference is to a system standard file or standard unit file (such as $OUTPUT or OUTPUT), it is assumed to be in the $LOCAL catalog.

If omitted, the name is derived from the UNIT= specifier, which is assumed to be in the working catalog.

stat

Character expression specifying file status. Valid values are:

'OLD'

File currently exists.

'NEW'

File does not currently exist.

'SCRATCH'

Delete the file associated with unit **u** upon program termination or execution of CLOSE that specifies unit **u**; it must not appear if FILE parameter is specified.

'UNKNOWN'

File status is unknown.

If omitted, STATUS='UNKNOWN' is used.

acc

Character expression specifying the access method for the file:

'SEQUENTIAL'

Open the file for sequential access.

'DIRECT'

Open the file for direct access.

If omitted, ACCESS='SEQUENTIAL' is used.

If the file exists when the OPEN is issued, the access method must be valid for that file. (The file must have been created with the specified access method.)

fm

Character expression specifying formatting option:

'FORMATTED'

Open the file for formatted input/output.

'UNFORMATTED'

Open the file for unformatted input/output.

'BUFFERED'

Open the file for buffered input/output.

If omitted, FORM='FORMATTED' is used for sequential access files, FORM='UNFORMATTED' is used for direct access files.

For an existing file, the specified form must be valid for that file. This specifier is not valid for segment access files.

rl

Positive integer variable or constant specifying the maximum record length for a direct or sequential access file. This specifier is required for a direct access file. If omitted for a sequential access file, 65535 characters is used.

This specifier is not valid for segment access files.

blnk

Character expression specifying how blank are interpreted for formatted input/output:

'NULL'

Blank values in numeric formatted input fields are ignored, except that a field of all blanks is treated as zero.

'ZERO'

Blanks, other than leading blanks are treated as zeros.

If omitted, BLANK='NULL' is used.

This specifier is not valid for segment access files.

bl

This parameter is provided only for compatibility with previous versions of FORTRAN and is disregarded.

Example:

```
OPEN (UNIT=2, FILE='AAA', ACCESS='DIRECT', RECL=120)
```

This statement opens file AAA in the working catalog for direct access input/output and associates AAA with unit 2.

A file specified in an OPEN statement is opened with the attributes specified in the OPEN statement. A file must be opened before any input/output operation can be performed on it. However, it is not always necessary to specify the file in an OPEN statement. If a unit is referenced in an input/output operation, and the file associated with that unit was not previously declared in an OPEN statement, an implicit OPEN of the file associated with the referenced unit occurs.

Files opened implicitly are assigned default attributes when the file is first referenced in an input/output statement. Files opened implicitly without a file path specification are assumed to be in the working catalog.

The UNIT= specifier is required in an OPEN statement; all other specifiers are optional except that you must specify the RECL specifier if a file is being opened for direct access and you must specify the FILE specifier to open a segment access file. If a the STATUS parameter specifies NEW or OLD, you must also specify the FILE parameter.

If you omit the FILE= specifier, the file is assumed to be the one associated with the specified unit in the PROGRAM statement (described in chapter 8, Program Units). If you do not specify the unit on the PROGRAM statement, the file name is derived from the unit number. For unit numbers in the range 0 through 999, the file name is TAPEn where *n* is the unit number; for unit numbers having the form of a 1- through 7-character name, the file name is the same as the unit number. The file is assumed to be in the working catalog.

A declaration in an OPEN statement overrides a declaration in a preceding PROGRAM statement, providing that no input/output operations have been performed on the file. For example, in the sequence

```
PROGRAM XX (TAPE2=/500)
        :
OPEN (2, RECL=1300, FILE='FILEY')
READ (2, 100) A, B, C
```

the PROGRAM statement declares a 500 character record length for unit 2; however, the OPEN statement takes precedence over the PROGRAM statement, and the 1300 character record length is used. The READ statement reads data from FILEY in the working catalog.

Files opened with STATUS='SCRATCH' are set to the $LOCAL catalog, regardless of the the existing working catalog.

Declarations of file properties on a SET_FILE_ATTRIBUTES command override any conflicting OPEN statement parameters for a unit associated with that file; this applies to all OPEN statements for that file. For example, a MAXIMUM_RECORD_LENGTH parameter on a SET_FILE_ATTRIBUTES command overrides the RECL parameter value specified in an OPEN statement.

When a file is created (opened for the first time), the attributes established by the OPEN statement are permanent. A subsequent open of the file uses the original attributes. The only specifiers that can be changed are the BLANK= and UNIT= specifiers. In order to define file attributes, you must specify all of the desired attributes in the first OPEN statement (or in a SET_FILE_ATTRIBUTE command prior to execution).

Once a file has been associated with a particular unit, the file can be associated with another unit in a subsequent OPEN statement. The file is then associated with more than one unit. In this case the unit numbers refer to the same file. Actions taken on one unit also affect the other unit. For example, closing a unit closes all other units associated with the same file.

Example:

```
OPEN (2, FILE='INFIL')
      :
OPEN (3, FILE='INFIL')
READ (2, 100) A, B
READ (3, 100) X, Y
```

Both READ statements read from file INFIL in the working catalog.

Example:

```
OPEN (3, FILE='$LOCAL.XXX', STATUS='OLD', BLANK='ZERO')
```

When data is read from the existing local file XXX, blanks are interpreted as zeros.

Example:

```
OPEN (2, STATUS='NEW', ERR=12, FILE='$USER.NEWFL', ACCESS='SEQUENTIAL')
```

A new file, $USER.NEWFL, is associated with unit 2 and is to be a sequential access file.

If a file is associated with a unit and a succeeding OPEN statement associates a different file with the same unit, the effect is the same as performing a CLOSE without a STATUS= specifier on the currently associated file before associating the new file with the unit. For example, in the sequence

```
OPEN (2, FILE='MYFILE')
WRITE (2, '(A)') A, B, C
OPEN (2, FILE='PART2')
```

the second OPEN statement implicitly closes MYFILE before opening PART2. Both files are assumed to be in the working catalog.

When opening a standard system file, such as $INPUT or $OUTPUT, you must specify STATUS='UNKNOWN' (or omit the STATUS= specifier since the default is also 'UNKNOWN'). Standard system files are assumed to be in the $LOCAL catalog.

To create and open a segment access file, specify the common block name as the UNIT specifier and the segment access file name as the FILE specifier (the FILE specifier is required for segment access files). For example:

```
      COMMON /CHBLK/ CH
C$    SEGFILE (/CHBLK/)
      OPEN (UNIT=/CHBLK/, FILE='$USER.SEG_FILE_1', ERR=10, IOSTAT=IVAR)
```

This creates and opens file $USER.SEG_FILE_1 and associates the file with the COMMON block CHBLK.

CLOSE

The CLOSE statement disassociates a file from a specified unit and specifies whether the file associated with that unit is to be kept or released. The CLOSE statement has the form:

CLOSE (*UNIT=* **u** , *IOSTAT=ios, ERR=sl, STATUS=sta, SIZE=n*)

u

Unit identifier of the file to be closed.

ios

Specifies an 8-byte integer variable or array element into which one of the following values is returned after the input/output operation is complete:

Value Returned	Status of Input/Output Operation
< 0	End-of-file encountered.
= 0	Operation completed normally.
> 0	An error occurred. The value is either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier and its condition number.

sl

Label of an executable statement to which control transfers if an error occurs during the close.

sta

Character expression that specifies the disposition of the file after the close:

'KEEP'

The file is kept after the close.

'DELETE'

The file is deleted after the close.

If omitted, STATUS='DELETE' is used if file status is 'SCRATCH'; otherwise, STATUS='KEEP' is used.

'KEEP' is not valid option for a file whose status is 'SCRATCH'.

n

An 8-byte integer expression specifying the length of a segment access file in bytes. This specifier is used only with segment access files.

If omitted, the size of the segment access file before an association was made is used. This size is extended if a reference is made beyond the current size (which is usually be the case if the file is being created).

A CLOSE statement can appear in any program unit in the program; it need not appear in the same program unit as the OPEN statement specifying the same unit.

A CLOSE statement that references a unit having no file associated with it has no effect.

A CLOSE statement disassociates a unit and dissolves all connections established by the PROGRAM statement and the LGO command. You can associate it again within the same program to the same file or to a different file. A file associated with a unit specified in a CLOSE statement can be subsequently associated with the same unit or another unit, provided the file still exists.

Unit equivalence established on the PROGRAM statement and file association established on the execution command are no longer in effect after the CLOSE statement is executed.

When a program terminates normally, the implicit statement

```
CLOSE (u, STATUS='KEEP')
```

occurs for each opened unit unless the status of the file was SCRATCH; in this case, the following implicit statement occurs:

```
CLOSE (u, STATUS='DELETE')
```

Do not specify STATUS='DELETE' when closing a standard system file, such as $INPUT or $OUTPUT. A standard system file cannot be deleted from the job.

Example:

```
CLOSE (2, ERR=25, STATUS='DELETE')
```

INQUIRE

The INQUIRE statement returns information about a specified file or unit. This statement has the form:

INQUIRE(unfl, *IOSTAT=ios, ERR=sl, EXIST=ex, OPENED=od, NUMBER=num, NAMED=nmd, NAME=fn, ACCESS=acc, SEQUENTIAL=seq, DIRECT=dir, FORM=fm, FORMATTED=fmt, UNFORMATTED=unf, REC=rcl, NEXTREC=nr, BLANK=blnk)*

unfl

Specifies the file or unit for which information is to be returned; unfl has one of the following forms:

UNIT=**u**

Inquire by unit; u is a unit identifier. For a segment access file, the unit identifier specifies the named common block (including slashes).

FILE=fin

Inquire by file; fin is a character expression whose value is a file reference.

ios

Specifies an 8-byte integer variable or array element into which one of the following values is returned after the input/output operation is complete:

Value Returned	Status of Input/Output Operation
< 0	End-of-file encountered.
= 0	Operation completed normally.
> 0	An error occurred. The value is either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier and its condition number.

sl

Label of an executable statement to which control passes if an error occurs during an inquire.

ex

An 8-byte logical variable or array element that receives a value indicating whether the file or unit is valid:

.TRUE.

For inquire-by-unit, the unit is a valid unit. For inquire-by-file, the file is local to the job and contains data.

.FALSE.

The file or unit is not valid.

od

An 8-byte logical variable or array element that receives one of the following values:

.TRUE.

The file (unit) is associated with a unit (file).

.FALSE.

The file (unit) is not associated with a unit (file).

num

An 8-byte integer variable or array element that receives the unit number of the unit currently associated with the file; undefined if the file is not associated with a unit.

nmd

An 8-byte logical variable or array element that receives one of the following values:

.TRUE.

The file has a name.

.FALSE.

The file does not have a name.

fn

A character variable or array element that receives the file reference or value of the file associated with unit **u**. The maximum size of a file reference is 512 characters.

An absolute file reference uses an absolute path, such as .ACCOUNT1.FILEA, rather than a relative path, such as $USER.FILEA. If fn is not long enough to hold the absolute file reference, an internally assigned 31-character file name is returned which can be used with an OPEN statement. If fn is not large enough to hold the returned file name, an error occurs, and nothing is returned in the fn parameter. You can use the INQUIRE statement to determine the internally assigned file name.

acc

A character variable or array element that receives a value indicating the access method of the file:

'SEQUENTIAL'

The file is opened for sequential access input/output.

'DIRECT'

The file is opened for direct access input/output.

If the file is not open, acc is undefined.

seq

A character variable or array element indicating whether the file can be opened for sequential access input/output:

'YES'

The file can be opened for sequential access input/output.

'NO'

The file cannot be opened for sequential access input/output.

'UNKNOWN'

It cannot be determined if the file can be opened for sequential access input/output.

dir

A character variable or array element indicating whether the file can be opened for direct access input/output:

'YES'

The file can be opened for direct access input/output.

'NO'

The file cannot be opened for direct access input/output.

'UNKNOWN'

It cannot be determined if the file can be opened for direct access input/output.

fm

A character variable or array element indicating whether the file is opened for formatted or unformatted input/output:

'FORMATTED'

The file is open for formatted input/output.

'UNFORMATTED'

The file is open for unformatted input/output.

'BUFFERED'

The file is open for buffered input/output.

If the file is not open, fm is undefined.

fmt

A character variable or array element indicating whether the file can be opened for formatted input/output:

'YES'

The file can be opened for formatted input/output.

'NO'

The file cannot be opened for formatted input/output.

'UNKNOWN'

It cannot be determined if the file can be opened for formatted input/output.

unf

A character variable or array element indicating whether the file can be opened for unformatted input/output:

'YES'

The file can be opened for unformatted input/output.

'NO'

The file cannot be opened for unformatted input/output.

'UNKNOWN'

It cannot be determined if the file can be opened for unformatted input/output.

rcl

An 8-byte integer variable or array element that receives the record length of a file opened for direct access. If the file is 'FORMATTED', rcl contains the record length in characters; if 'UNFORMATTED', the record length is in words. This parameter is undefined if the file is not opened for direct access.

nr

An 8-byte integer variable or array element; for a direct access file, nr receives the record number of the next record to be read or written. If no records have been read or written, 1 is returned. This parameter is undefined for sequential access files.

blnk

A character variable or array element indicating blank interpretation currently in effect for a file opened for formatted input/output:

'NULL'

Null blank control is in effect.

'ZERO'

Zero blank control is in effect.

If the file is not opened for formatted input/output, blnk is undefined.

The two forms of the INQUIRE statement are inquire by unit and inquire by file. Inquire by unit is used to obtain information about the current status of a specified unit; inquire by file is used to obtain information about the current status of a file.

If a common block name is used for an inquire by unit or if a segment access file is specified on an inquire by file, then only the IOSTAT, ERR, EXIST, OPENED, NAMED and NAME specifiers are valid.

You must specify either a file reference (inquire by file) or a unit specifier (inquire by unit), but not both, in an INQUIRE statement. The file or unit need not exist when INQUIRE is executed. Following execution of an INQUIRE statement, the specified parameters contain values that are current at the time the statement is executed. If a unit number is specified and the unit is opened, the following specifiers contain information about the file associated with the unit (provided that any conditions stated in the specifier descriptions hold):

NAMED	NAME	FORM
SEQUENTIAL	DIRECT	RECL
FORMATTED	UNFORMATTED	EXIST
OPENED	NEXTREC	BLANK
NUMBER	ACCESS	

If a file name is specified, the following parameters contain information about the file and the unit with which it is associated (provided that any conditions stated in the specifier descriptions hold):

NAMED	NAME	SEQUENTIAL
DIRECT	FORMATTED	UNFORMATTED
OPENED	EXIST	NUMBER
ACCESS	FORM	RECL
NEXTREC	BLANK	

On an inquire by unit for a common block name, EXIST is always true and NAMED is the same as OPENED. On an inquire by file for a segment access file, NAME returns the common block name (including slashes) and NAMED is always true.

If the internally-assigned file is too large for the FILE parameter specification, the error condition is FLE$NAME_VAR_TOO_SHORT_FOR_FILE is returned. To test for this condition, you can use the CONDNAM function with the value returned in the iostat parameter. The CONDNAM function is described under NOS/VE Status Subprograms in chapter 10.

The RECL value returned is zero for an inquire by unit on an unopened but existing file.

If a file is specified that is associated with more than one unit, the NUMBER parameter contains one of the unit numbers or names.

If you specify an invalid file or unit, no error results, but certain parameters are not assigned values. If you specify a unit that is not associated with a file, only the OPENED, IOSTAT and EXIST specifiers receive values.

If an error occurs during an INQUIRE, only the IOSTAT specifier contains a value.

Example:

```
LOGICAL EX
CHARACTER*10 AC
      :
INQUIRE (FILE='AFILE', ERR=100, EXIST=EX, ACCESS=AC)
```

Status information for file AFILE is returned in the variables EX and AC. If an error occurs during the INQUIRE, control transfers to statement 100.

File Positioning Statements

Three statements are provided to position files opened for sequential access: REWIND, BACKSPACE, and ENDFILE. You cannot use these statements on files opened for direct access.

The BACKSPACE, REWIND, and ENDFILE operations are valid only for sequential files with V type records. BACKSPACE skips backward (toward beginning-of-information) one record. (The file is positioned before the record just read or written). REWIND positions a file at its beginning-of-information. ENDFILE writes an end-of-partition boundary.

REWIND

The REWIND statement positions a sequential access file at its beginning-of-information. This statement has the form:

REWIND(*UNIT*=**u**, *IOSTAT=ios, ERR=sl***)**

REWIND u

u

External unit identifier. The characters UNIT= are optional.

ios

An 8-byte integer variable which returns an error number; a value of 0 indicates no errors occurred.

sl

Label of an executable statement to which control transfers if an error occurs during the rewind.

The next input/output operation after a rewind references the first record in the file. If the file is already at its beginning-of-information, no action is taken.

Example:

```
REWIND 3
```

The file associated with unit 3 is positioned at beginning-of-information.

BACKSPACE

The BACKSPACE statement positions the sequential access file associated with the specified unit one record in a backward direction (toward its beginning-of-information). This statement has the form:

BACKSPACE(*UNIT*=u, *IOSTAT=ios, ERR=sl*)

BACKSPACE u

u

External unit identifier. The characters UNIT= are optional.

ios

An 8-byte integer variable which returns an error number; a value of 0 indicates no errors occurred.

sl

Label of an executable statement to which control transfers if an error occurs during the backspace.

If the file is already positioned at beginning-of-information, no action is taken. You cannot not use backspace operations on records created by list directed or namelist output.

Example:

```
      DO 1 LUN=1, 4
1     BACKSPACE LUN
```

The files associated with units 1 through 4 are backspaced one record.

ENDFILE

The ENDFILE statement writes an end-of-partition boundary on the sequential access file associated with the specified unit. This statement has the form:

ENDFILE (*UNIT=* **u** , *IOSTAT=ios, ERR=sl*)

ENDFILE u

u

External unit identifier. The characters UNIT= are optional.

ios

An 8-byte integer variable which returns an error number; a value of 0 indicates no errors occurred.

sl

Label of an executable statement to which control transfers if an error occurs during the endfile.

The end-of-partition can be detected by the END= and IOSTAT= specifiers.

The following restrictions apply to ENDFILE:

- ENDFILE should not be the first operation on a file.
- ENDFILE is not permitted on units opened for direct access.
- ENDFILE cannot be used on a file processed by the mass storage subroutines.
- ENDFILE can only be used on files with V- type records.

ENDFILE can be used to flush terminal output unconditionally. If the unit specified is associated with a file that is connected to a terminal, the output buffer for the file is flushed to display messages immediately.

Example:

```
IOUT=7
ENDFILE (UNIT=IOUT, ERR=100)
```

An end-of-partition boundary is written on the file associated with unit 7.

Control Data Extension

Input/Output Status Checking Functions

Status checking for input/output statements such as READ and WRITE should be done with the optional specifiers IOSTAT= or END= , but can also be done with the functions UNIT, EOF, and IOCHEC. UNIT and EOF return an end-of-file status if an end-of-partition or end-of-information was read by the previous read operation.

For F and U type records, the EOF and UNIT functions return end-of-file status only at end-of-information because these record types do not support position boundaries.

The functions UNIT and IOCHEC return a parity error indication for every record within or spanning a block containing a parity error; such an indication, however, does not necessarily refer to the immediately preceding operation because of the record blocking/deblocking performed by the internal input/output routines.

NOTE

All arguments to the input/output status checking routines must be 8-byte values.

UNIT

The UNIT function checks the status of a BUFFER IN or BUFFER OUT operation for an end-of-file or parity error condition on logical unit u. The UNIT function reference has the form:

UNIT(u, a, b)

u

Unit identifier.

a

First variable or array element of the block of memory specified in the preceding BUFFER IN or BUFFER OUT statement.

b

Last variable or array element of the block of memory specified in the preceding BUFFER IN or BUFFER OUT statement.

The function returns one of the following type real values:

−1. Unit ready, no end-of-file or parity error encountered on the previous operation.

0. Unit ready, end-of-file encountered on the previous operation.

+1. Unit ready, parity error encountered on the previous operation.

Control Data Extension *(Continued)*

You should always include **a** and **b** if you have selected OPTIMIZATION_ LEVEL=HIGH on the VECTOR_FORTRAN command. Specifying **a** and **b** allows the FORTRAN compiler to associate the call to UNIT with possible changes to the values in the locations between a and b. If you call UNIT with only the argument u (as in most older programs), the compiler may not detect that values between a and b are being referenced while instructions between the BUFFER IN or BUFFER OUT statement and the UNIT call are executing. This could lead to unpredicatable results.

Example:

```
BUFFER IN(5, 1) (B(1), B(100))
IF(UNIT(5, B(1), B(100)) 12, 14, 16
```

Control transfers to the statement labeled 12, 14, or 16 if the value returned was -1., 0., or +1., respectively.

If 0. or +1. is returned, the condition indicator is cleared before control is returned to the program. UNIT should be called only for a file processed by BUFFER statements.

EOF

The EOF function testa for an end-of-file on a unit following a formatted, list directed, namelist, or unformatted sequential read. The EOF function reference has the form:

EOF(u)

u

Unit identifier.

Zero is returned if no end-of-file is encountered; a nonzero value is returned if an end-of-file is encountered. If an end-of-file is encountered, EOF clears the condition indicator before returning control.

Example:

```
IF(EOF(5) .NE. 0) GO TO 20
```

Control transfers to statement 20 if an end-of-file is encountered on unit 5.

The EOF function is provided for compatibility with previous systems and is not intended to replace the END= or IOSTAT= specifiers in a READ statement. The IOSTAT= or END= specifier must be used in a READ statement that reads an end-of-file.

The EOF function should not test for an endfile condition following read or write operations on random access files (files accessed by READMS/WRITMS) or following write operations on all types of files, because a zero value is always returned regardless of whether an end-of-file is detected.

The EOF function is of type real.

Control Data Extension *(Continued)*

IOCHEC

The IOCHEC function tests for a parity error on a unit following a formatted, list directed, namelist, or unformatted read. The IOCHEC function reference has the form:

IOCHEC(u)

u

Unit identifier.

Zero is returned if no error was detected. If a parity error occurs, IOCHEC clears the parity indicator before returning control.

The IOCHEC function is of type integer.

Example:

```
    READ(UNIT=6, END=99, ERR=88) A
88  J=IOCHEC(6)
    IF(J .NE. 0) GO TO 25
```

If no parity error occurs during the READ (IOCHEC returns zero), execution continues with the statement following the IF. If a parity error is detected, control transfers to statement 25.

LENGTH and LENGTHX

The LENGTH function and the LENGTHX subroutine return the number of words in the last record read by the previous formatted READ, list directed READ, BUFFER IN, or READMS of the specified unit. The LENGTH reference and LENGTHX call have the forms:

LENGTH(u)

CALL LENGTHX(u, nw, ubc)

u

Unit identifier.

nw

Integer variable or array element to receive the number of words read.

ubc

Integer variable or array element to receive the number of unused bits in the last word of the transfer.

The values returned by LENGTHX and the value of LENGTH are of type integer.

For a file accessed by buffer statements, LENGTH or LENGTHX should be called only after a call to UNIT ensures that input/output activity is complete; otherwise, file integrity might be endangered.

A length of zero is returned if the previous READ was NAMELIST.

Control Data Extension *(Continued)*

Example:

```
DIMENSION CALC(51)
BUFFER IN(1, K) (CALC(1), CALC(51))
J=LENGTH(1)
IF(UNIT(1) .GE. 0) GO TO 20
```

The variable J contains the number of words read on unit number 1.

LENGTHB

The LENGTHB function returns the number of bytes in the last record read by the previous formatted READ, list directed READ, BUFFER IN, or READMS of the specified unit. The LENGTHB function has the form:

LENGTHB(u)

u

Unit identifier.

The value returned by LENGTHB is of type integer.

A length of zero is returned if the unit is not open, or if the unit is open but no input activity has been previously completed on that unit.

A length of zero is also returned if the previous READ was NAMELIST.

Example:

```
DIMENSION CALC(51)
BUFFER IN(1, K) (CALC(1), CALC(51))
J=LENGTHB(1)
IF(UNIT(1) .GE. 0) GO TO 20
```

The variable J contains the number of bytes read on unit number 1.

Internal Data Transfer Routines

Two routines are provided for moving blocks of data from one area of memory to another: MOVLEV and MOVLCH.

For Better Performance

The Internal Data Transfer Routines can increase execution time and are provided only for compatibility with other versions of FORTRAN.

MOVLEV

The MOVLEV call transfers blocks of noncharacter data from one area of memory to another. This call has the form:

CALL MOVLEV(a, b, n)

a

Variable or array element indicating the starting location of the data to be moved. Integer or logical data must be 8 bytes in length.

b

Variable or array element indicating the starting location of the area to receive the data. Integer or logical data must be 8 bytes in length.

n

Integer expression specifying the number of words to be moved.

No conversion is done by MOVLEV. For instance, if data from a real variable is moved to a type integer receiving field, the result is undefined if referenced as an integer variable or array element.

The block of data to be moved should be contained in one variable, array, equivalence class, or common block, and the receiving area should be contained in one variable, array, equivalence class, or common block. Otherwise, the result of the move is undefined. Also, the blocks should not overlap.

The arguments a and b must not be type character; for character data, subroutine MOVLCH should be used.

Example:

```
CALL MOVLEV(A, I, 1000)
```

This call moves 1000 words from locations starting with A to locations starting with I. No conversion is performed.

Example:

```
DOUBLE PRECISION D1(500), D2(500)
CALL MOVLEV(D1, D2, 1000)
```

Because array D1 is defined as double precision, the word count is set to twice the size of D1 so that the entire array is moved.

Control Data Extension *(Continued)*

MOVLCH

The MOVLCH call transfers character data from one area of memory to another. This call has the form:

CALL MOVLCH(a, b, n)

a

Variable, array element, or substring name indicating the starting location of the character string to be moved.

b

Variable, array element, or substring name indicating the starting location of the receiving area.

n

Integer expression specifying the number of characters to be moved.

The block of data to be moved should be contained in one variable, array, equivalence class, or common block, and the receiving area should be contained in one variable, array, equivalence class, or common block. Otherwise, the result of the move is undefined. Also, the blocks should not overlap.

Both arguments a and b must be of type character; a diagnostic is issued if one, or both, is of any other type. Note that moving a single character variable or array element to another single character variable or array element is easier to do with a character assignment statement.

Example:

```
CHARACTER*123, CH1(10), CH2(5)
CALL MOVLCH(CH1(8), CH2(3), 369)
```

The last three elements of character array CH1 are moved to the last three elements of character array CH2. Because each element is 123 characters long, the character count is set to 3 * 123=369.

File Connection Routines

Two routines are provided for connecting and disconnecting a file from the terminal.

CONNEC

The CONNEC call connects a $LOCAL file to the terminal with terminal attributes appropriate for the designated character set. This call has the form:

CALL CONNEC(u, *cs***)**

u

Unit identifier.

cs

Character set designator; an 8-byte integer expression having one of the following values:

0 Selects the ASCII-128 character set. The terminal attributes currently in effect are used.

1 Same as 0, except that if transparent mode was on, it is turned off and associated transparent mode attributes are reset to their default values.

2 Selects the ASCII-256 character set. Transparent mode is turned on so that all bits of each byte (including control characters) can be transmitted as data provided your terminal uses 8 bits with no parity.

If cs is omitted, the character set defaults to ASCII-128. The ASCII 128-character set is shown in appendix C.

If a program to be run interactively calls for input/output operations through a remote terminal, all files to be accessed through the terminal must be formally associated with the terminal when the files are referenced.

Files $INPUT and $OUTPUT are automatically connected to the terminal. (However, you can override this connection.) Thus, whenever data is read from file $INPUT, the data must be entered through the terminal. Whenever data is written to file $OUTPUT, the data is displayed at the terminal.

You can connect any file from within the program by using the CALL CONNEC statement. You can also connect a file by specifying a REQUEST_TERMINAL command (described in the NOS/VE System Usage manual).

If a unit specified in a CONNEC call is associated with a file that is open but is not connected to a terminal, the file is first closed (and its buffer flushed) and then connected to the terminal under the specified character set option. This procedure is also followed for files connected using the REQUEST_TERMINAL command.

If a unit specified in a CONNEC call is already connected to the terminal and is also open, the terminal attributes are set according to the specified option; the associated file is not closed.

Control Data Extension *(Continued)*

DISCON

The DISCON call disconnects a file from the terminal. This call has the form:

CALL DISCON(u)

u

Unit identifier.

The file associated with the specified unit is detached, and data written to the file is lost. A DISCON call is ignored if the file is not connected.

Formatted Data Mismatch

Mismatches between actual data types and types specified in a FORMAT statement normally cause an error. To ignore such mismatches at compile time, you can use the CALL IGNFDM statement.

IGNFDM

The IGNFDM call can be used to ignore data type mismatches in formatted input/output statements. This call has the form:

CALL IGNFDM (ignore_flag)

ignore_flag

Logical expression specifying true if mismatches are to be ignored or false if mismatches are to be recognized. If omitted, ignore_flag is false.

If the value of ignore_flag is true, then a mismatch between variable type of non-character data and non-A type edit descriptors for formatted input/output statements, is ignored. If the value of ignore_flag is false, a mismatch between variable type and edit descriptors in formatted input/output statements causes a runtime error.

When the value of ignore_flag is true, intermixed real and integer data and edit descriptor mismatches cause no errors. However, trying to output an integer with a double-precision edit descriptor might cause an error.

Only one call is needed to enable this feature for execution of a particular program.

Example:

```
      INTEGER N
      CALL IGNFDM(.TRUE.)
      READ (5, 100) N
100   FORMAT (F10.2)
```

The call to IGNFDM allows the variable N, which is typed as integer, to receive a floating-point number. While this causes no error, it could cause unpredictable results if N is later referenced as an integer.

End of Control Data Extension

Control Data Extension

FORTRAN Fast I/O

FORTRAN fast I/O improves internal processing of buffered, direct-access, and sequential input/output for certain files. The files are those that can be opened with access modes of READ and WRITE, share modes of NONE, and meet the following criteria:

- The file is not a system standard file
- The file is either fixed or variable record type
- The file is a system-specified block type
- The file has no associated file-access procedure
- The file has a blank padding character.

Fast I/O is not used for terminal or tape files. When a file is opened for fast I/O, the file cannot be opened again concurrently. It is necessary to close the file first before it can be opened again. This can cause differences in some input/output situations. The following paragraphs describe the types of differences that occur with fast I/O and ways to turn off fast I/O.

Open Sharing

Some programs that want to share opens may behave differently due to fast I/O. The message, *open share mode NONE* means that a file cannot be opened again by NOS/VE or another language or utility (such as COBOL or SORT), until it is first closed. In some cases, a FORTRAN OPEN statement within the same task can be performed on a file that FORTRAN has already opened. For example, a FORTRAN OPEN satement can change the BLANK= specifier that applies from a previous OPEN statement. This is not really opening the file again so it can be done without the file being sharable.

If a file must be shared, and its attributes do not preclude fast I/O, the user must prohibit fast I/O on the file by turning off fast I/O or by using one of the methods described in the following paragraphs to turn off fast I/O for individual files.

Files Shared With Another Task or Another Language Subprogram

If another task, or another language subprogram (such as COBOL), tries to do input/output operations on a file that is opened for fast I/O, the routine will fail because other languages try to open the file again. You must close the FORTRAN file before calling another language subprogram.

Control Data Extension *(Continued)*

Connected Files

Programs that write to a file in more than one instance of open may behave differently using fast I/O. For example, assume the following file connection exists:

```
/create_file_connection  $errors  tape6
```

If a FORTRAN program opens TAPE6 for fast I/O, then a later attempt to write to $ERRORS causes incorrect information to be written to $ERRORS. This error occurs because TAPE6 cannot be opened once it is already opened for fast I/O. This problem can be avoided by using the commands:

```
/create_file_connection  $errors  tape6x
/create_file_connection  tape6  tape6x
```

(Neither TAPE6 or $ERRORS is a disk file.)

Programs With CALL SCLCMD Statements

If a file is open for fast I/O, a CALL SCLCMD statement to write onto the same file aborts because the NOS/VE command attempts to open the file. For example, if TAPE6 is being used by a FORTRAN program and has been opened for fast I/O, then

```
CALL SCLCMD ('COPY_FILE F TAPE6.$EOI' )
```

causes the program to abort.

This can be avoided by using the following statement before the call to SCLCMD:

```
CLOSE (6, STATUS = 'KEEP')
```

Fast I/O for Individual Files

Fast I/O is automatically used on files residing in the $LOCAL catalog. If you want to prevent this, make the files permanent with the appropriate access and share modes. For example,

```
/create_file  $user.tape6
/detach_file  $user.tape6
/attach_file  $user.tape6  share_modes=none
```

Fast I/O will not be used on permanent file $user.tape6 because the file has the default ATTACH_FILE attributes of READ and EXECUTE, but not of WRITE which is required for fast I/O.

Fast I/O is also not used when individual files are attached with share modes other than none.

Control Data Extension ***(Continued)***

How to Turn Off Fast I/O

To turn off fast I/O, you can do one of the following:

- Create an SCL variable accessible to the FORTRAN program. The variable must be named FLV$IO_OPT_HIGH, of kind boolean and an initial value of NO (or FALSE or OFF):

  ```
  /var
  var/flv$io_opt_high; boolean  value=no  scope=job
  var/varend
  ```

  When the value of this variable is NO (at the beginning of a FORTRAN program) no file is opened for fast I/O. Fast I/O can be activated again:

  ```
  /flv$io_opt_high=on
  ```

  This variable has no effect at compile time.

- Set the pad character of a file to any character other than blank.

- Use a block type other than system-specified or a record type other than fixed or variable. This method may increase execution time.

- Connect a file to another file or associate a file access procedure to a file. This method may increase execution time.

End of Control Data Extension

Program Units 8

Overview of Program Units and Statement Functions 8-2

Main Programs 8-4
  PROGRAM Statement 8-5

Subprograms 8-7
  Subroutine Subprogram 8-8
    How to Call a Subroutine Subprogram 8-9
  Function Subprogram 8-12
    External Functions 8-12
    How to Reference an External Function 8-14
  Block Data Subprogram 8-16

Statement Functions 8-17
  How to Reference a Statement Function 8-18

How Program Units Pass Values 8-20
  Argument Lists 8-21
    Actual Arguments 8-21
    Dummy Arguments 8-22
    How to Associate Actual and Dummy Arguments 8-23
    Variables as Arguments 8-25
    Subprogram Names as Arguments 8-25
    Asterisks as Arguments 8-25
    Arrays as Arguments 8-26
    Adjustable Dimensions 8-29
      Assumed-Size Arrays 8-29
      Assumed-Shape Arrays 8-30
    Character Values as Arguments 8-31
      Assumed-Length Character Strings 8-32
  Common Blocks 8-33
  ENTRY and RETURN Statements 8-34
    ENTRY statement 8-34
    RETURN Statement 8-36

Program Units 8

An executable FORTRAN program consists of one or more program units. One of the program units must be a main program unit, usually called a main program.

This chapter describes program units and statement functions in the following order:

- Main programs
- Subprograms
  - Subroutines
  - Functions
  - Block data
  - Statement functions

This chapter also describes how program units and statement functions communicate:

- Arguments
- Common blocks

This chapter also describes two statements used to communicate between program units:

- The ENTRY statement
- The RETURN statement

Overview of Program Units and Statement Functions

A program unit is a group of FORTRAN statements, with optional comments, terminated by an END statement. The types of program units are main programs and subprograms. Subprograms can be subroutine subprograms, function subprograms, or block data subprograms. Statement functions are similar to function subprograms, although they are not program units.

A function subprogram can be an external function or an intrinsic function. Intrinsic functions are supplied by the Math library and can be referenced by any FORTRAN program. The intrinsic functions are described in chapter 9. External functions are written by you.

Statement functions are defined within program units and are compiled within the program unit. (Statement functions cannot be compiled separately.)

The following table summarizes the characteristics of program units and statement functions:

Program Unit	Characteristics	How Identified
Main program	Defines a main entry point for a FORTRAN program. Every program must have a main program unit.	Usually begins with PROGRAM statement.
Subroutine	Can be called from other program units in a program. Returns values through argument list or common.	Begins with SUBROUTINE statement.
External function	Can be called from other program units. Returns value through function name. Can also communicate through argument list and common.	Begins with FUNCTION statement.
Block data subprogram	Provides initial values for named common blocks.	Begins with BLOCK DATA statement.
Statement functions	Calculates a single result for a program unit; cannot be referenced outside the defining program unit.	Defined within a program unit by a single statement.

Figure 8-1 illustrates a FORTRAN program consisting of three program units: a main program named AVG, a subroutine subprogram named DIVIDE, and a function subprogram named NADD.

```
      PROGRAM AVERAGE
      INTEGER NUMBER(10), NSUM
      REAL RESULT
      OPEN(1, FILE='$USER.DATA')
      READ(1, 100) (NUMBER(I), I=1, 10)
      NSUM=NADD(NUMBER)
      CALL DIVIDE(NSUM, 10, RESULT)
      PRINT 200, (NUMBER(I), I=1, 10), RESULT
      STOP
 100  FORMAT(10I2)
 200  FORMAT('1THE AVERAGE OF ', /10(' ', I2), /, ' IS ', F7.3)
      END

      FUNCTION NADD(IARRAY)
      INTEGER IARRAY(10)
      N=0
      DO 10 I=1, 10
      N=IARRAY(I) + N
 10   CONTINUE
      NADD=N
      RETURN
      END

      SUBROUTINE DIVIDE (N, I, Q)
      INTEGER N, I
      REAL Q
      Q=REAL(N) / REAL(I)
      RETURN
      END
```

Figure 8-1. Main Program, Function, and Subroutine Example

Main Programs

A main program is a program unit that does not begin with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. Usually, a main program begins with a PROGRAM statement, but this statement is optional. Every FORTRAN program begins executing with the main program unit.

A main program can contain any FORTRAN statements except FUNCTION, SUBROUTINE, BLOCK DATA, or ENTRY. The main program should have a PROGRAM statement and at least one executable statement, and it must have an END statement as the last statement. An executable program should not have more than one main program unit. (If more than one main program exists, the last one loaded is used.)

The main program can be compiled independently of any subprograms. However, when a main program is loaded into memory for execution, all the required subprograms must also be loaded and ready for execution.

To compile and execute a main program, see chapter 12, Compilation and Execution.

Figure 8-1 shows an example of a main program. The program, named AVERAGE, reads numbers from file $USER.DATA and calls function NADD and subroutine DIVIDE to perform calculations.

PROGRAM Statement

The PROGRAM statement is the name used as the entry point name and as the object program name for the loader. The PROGRAM statement can also declare certain properties of input/output units to be used by the program. The PROGRAM statement has the forms:

PROGRAM *name*

PROGRAM *name* **(upar, ...,** *upar***)**

name

Program name. If omitted, START# is used.

upar[1]

Declares an input/output unit in one of the following forms:

unit

Name of a unit to be used by the main program or its subprograms; one through seven characters. Maximum number of units is 49.

unit=n

This form is provided for compatibility with previous versions of FORTRAN, and is interpreted as if **=n** had been omitted.

unit=/r

Specifies the maximum record length in characters for list directed, formatted, and namelist lines; default length is 65535 characters. Maximum value is 65535 characters.

unit=n/r

This form is provided for compatibility with previous versions of FORTRAN; **n** is disregarded and **r** specifies the maximum record length in characters for list directed, formatted, and namelist lines.

altunit=unit

Altunit and unit are unit names; altunit is usually in the form TAPEu, where u is an integer in the range 0 through 999. This form specifies that the unit names are equivalent. The record length for altunit is as previously specified (or defaulted) for the unit.

The program name must not be the same name as any other program unit, entry point, or common block name. Also, the name cannot be the same as any other symbolic name in the main program.

1. Unit declaration on the PROGRAM statement is provided for compatibility with previous versions of FORTRAN. The OPEN statement can also be used to declare input/output units, and it conforms to the ANSI standard.

Example:

`PROGRAM FIRST`	This statement assigns the name FIRST to the program.

Example:

`PROGRAM PROGA(AFILE, TAPE2=AFILE)`	This statement assigns the name PROGA to the program, and equivalences the name TAPE2 to AFILE. When any input/output statement in the program references unit 2, the reference applies to unit AFILE.

The PROGRAM statement can declare input/output units that are used in the program and in any subprograms that are called. You can also use an OPEN statement to declare input/output units. If a unit is not declared on the PROGRAM statement or in an OPEN statement, an implicit open occurs on the first reference to the unit.

In most cases, the OPEN statement is preferred over the PROGRAM statement's unit declarations because OPEN provides more options, more flexibility, and is consistent with ANSI FORTRAN.

In a file name is not specified in your program, FORTRAN adds the characters TAPE as a prefix to the unit number to form the file name. For instance, TAPE3 is the file name assigned to unit number 3 and TAPE5 is the file name assigned to unit number 5. TAPE5 and TAPE05 do not specify the same file name.

Subprograms

A subprogram is a program unit that is physically included only once in a FORTRAN program but can be executed many times. It can be called from the main program or from another subprogram.

A subprogram is subordinate to a main program; that is, a subprogram cannot be executed independently of a main program. It can be compiled independently of a main program. However, a subprogram can be executed only after the main program begins execution. You generally write subprograms to perform calculations that are required repeatedly by a program or different programs.

The types of subprograms are

- Subroutines
- Functions
- Block data

Subroutine Subprogram

A subroutine subprogram begins with a SUBROUTINE statement and ends with an END statement. The SUBROUTINE statement has the form:

SUBROUTINE name*(d, ..., d)*

name

Name of the subroutine subprogram. This is the main entry point of the subprogram.

d

Optional; dummy argument that can be a variable name, array name, dummy procedure name, or asterisk

The SUBROUTINE statement must appear as the first statement of the subroutine subprogram. The subroutine name must not be the same as any other program unit or entry name. Also, the name cannot be the same as any other symbolic name in the subroutine except a common block name. You can specify additional entry points using ENTRY statements (see ENTRY Statement in this chapter).

Examples:

```
SUBROUTINE GETVAL(VAR,RVAL,N,Z)
     :
RETURN
END

SUBROUTINE ARC
     :
RETURN
END
```

Subroutines can contain any statements except a PROGRAM, BLOCK DATA, FUNCTION, or another SUBROUTINE statement. When a RETURN or END statement is encountered, control returns to the calling program unit.

Subroutines can communicate with other program units through the argument list. The arguments specified in the subroutine argument list are called dummy arguments and are associated with actual arguments specified in the CALL statement that calls the subroutine. (See How to Call a Subroutine Subprogram in this chapter). Subroutines can also communicate with other program units through common blocks. (See Common Blocks in this chapter.)

How to Call a Subroutine Subprogram

A subroutine subprogram is called when a CALL statement naming the subroutine is encountered in a program unit. The program unit containing the CALL statement is knows as the calling program. The CALL statement has the form:

CALL name(*a*, ..., *a*)

name

Subroutine name; cannot be the same as a variable name in the calling program unit.

a

Optional; actual argument that can be one of the following:

A constant (including a symbolic constant or extended Hollerith constant)

A variable, array, array section, or array element name

An expression with operators (except for a character expression involving concatenation of a dummy argument with length (*)); the expression cannot be array-valued

An intrinsic function name (with a scalar argument)

An external subroutine or function name

A dummy subroutine or function name

An alternate return specifier of the form *sl, where sl is the label of an executable statement that appears in the calling program unit

The CALL statement can contain actual arguments, which must correspond in order, number, length (in bytes), and type to those in the subroutine dummy argument definition. An actual argument of type boolean can have a corresponding dummy argument of type integer or real. An actual argument of type integer (8 bytes) or real (8 bytes) can have a corresponding dummy argument of type boolean.

An actual argument in a CALL statement can be a dummy argument name that appears in the dummy argument list of the subprogram containing the subroutine call. An asterisk dummy argument cannot be used as an actual argument.

A CALL statement can be used to call subprograms written in languages other than FORTRAN. See appendix D, Calling Other Language Subprograms.

A subroutine must not directly or indirectly call itself.

Figure 8-1 (at the beginning of this chapter) shows an example of a subroutine subprogram. The example contains the following:

```
  :
CALL DIVIDE(NSUM, 10, RESULT)
  :
SUBROUTINE DIVIDE (N, I, Q)
INTEGER N, I
REAL Q
Q=REAL(N) / REAL(I)
RETURN
END
```

The subprogram, named DIVIDE, receives values through the argument list. The dummy arguments are N, I, and Q. When the subroutine is called in the main program by the CALL DIVIDE statement, the actual arguments NSUM, 10, and RESULT, are associated with the dummy arguments. The result of the subroutine is stored in the dummy argument Q which is associated with the actual argument RESULT in the main program.

For Better Performance

Subroutine calls and returns within a loop involve longer execution time. If possible, try to put the loop inside of the function or subroutine itself and call it only once. This way, the subroutine is only called once, rather than on each pass through the loop. For example:

```
      DIMENSION DATA(100)
      DO 20 J=1, 100
        CALL SUBA(DATA(J))
   20 CONTINUE
          :
      SUBROUTINE SUBA(VALUE)
      VALUE=VALUE**2
      RETURN
      END
```

can be changed to execute faster:

```
      DIMENSION DATA(100)
      CALL SUBA(DATA)
          :
      SUBROUTINE SUBA(VALUE)
      INTEGER VALUE(100)
      DO 20 J=1, 100
        VALUE(J)=VALUE(J)**2
   20 CONTINUE
      RETURN
```

Also, if the subprogram contains a RETURN statement that causes frequent immediate returns, modify the subprogram so the need for frequent returns is eliminated. For example, if the value of N is frequently 0,

```
CALL X(A, B, N)
     :
SUBROUTINE X(A, B, N)
IF(N.EQ.0) RETURN
     :
RETURN
END
```

can be changed to execute faster

```
IF(N.NE.0) THEN
    CALL X(A, B, N)
ENDIF
     :
SUBROUTINE X(A, B, N)
     :
RETURN
END
```

Function Subprogram

Functions can be external functions or intrinsic functions. All external and most intrinsic functions are external to the program unit that references them. External function subprograms are written by you. Intrinsic functions are supplied by the FORTRAN compiler and library and are described in chapter 9, Intrinsic Functions.

External Functions

An external function begins with a FUNCTION statement and ends with an END statement. The FUNCTION statement has the form:

type **FUNCTION name****len(d, ..., d)*

type

Options are INTEGER*len, REAL*len, DOUBLE PRECISION, COMPLEX*len, LOGICAL*len, BOOLEAN, BYTE, or CHARACTER*clen, where clen is the number of characters in the function result.

If *type* is omitted, the function name determines the type according to the rules described under Data Types of Variables in chapter 2, Language Elements.

name

Name of the function subprogram. This is the main entry point of the function.

len

Length of the result of the function. Options are 2, 4, and 8 for type integer functions; 8 and 16 for type real functions; 16 for type complex functions; and 1, 2, 4, and 8 for type logical functions.

d

Optional dummy argument that can be a variable name, array name, or dummy procedure name.

The END statement is described in chapter 6, Flow Control Statements.

A function name must not be the same as any other name, except a variable name or common block name.

You can specify additional entry points using ENTRY statements (see ENTRY Statement in this chapter).

A function subprogram returns a single value to the calling program unit through the function name. At some point within the function, an assignment statement must be executed that defines the function name (or an entry name of the same type as the function name). When control returns to the referencing program unit, the value for the function reference is the value that was assigned to the function name. The function name can be referenced as a variable or redefined later in the function.

A function must not directly or indirectly reference itself.

For Better Performance

Double precision and real*16 functions require more execution time because of the extra precision they support (two words). You should use them only when necessary.

If the result of a function is of type character with length (*), it cannot be used as an operand for concatenation except in a character assignment statement.

A function subprogram can also communicate with the referencing program unit through a list of arguments or through common blocks. (See Arguments or Common Blocks later in this chapter.)

Function subprograms can contain any statements except PROGRAM, BLOCK DATA, SUBROUTINE, or other FUNCTION statements. Control is returned to the referencing program unit when a RETURN or END is encountered; a RETURN statement of the form RETURN exp in a function subprogram is not allowed. The RETURN statement is described in this chapter.

The name of a function specified in a FUNCTION (or ENTRY) statement must not appear in any other nonexecutable statement, except a type statement. If the type of a function is specified in a FUNCTION statement, then the function name cannot appear in a type statement.

The type and length of the function must be the same in the referencing program unit and the function subprogram. In the absence of explicit typing, the type of the function is determined by the first character of the function name. Implicit typing by the IMPLICIT statement takes effect only when the function name is not explicitly typed. The name cannot have its type explicitly specified more than once.

If the function name is of type character, then each entry point name must be type character. The length declared for a type character function must be the same in the referencing program unit and in the referenced function. The function name and entry names must also have the same length. For example, if the function name has a length of (*), all entry names must have a length of (*).

Examples of FUNCTION statements:

```
FUNCTION FN(X)

INTEGER FUNCTION FSMALL*2

INTEGER FUNCTION DZ(V1, V2, V3)

CHARACTER*3 FUNCTION VCHARS(CH)

CHARACTER*(*) FUNCTION APPLE(TXT)
```

The last statement in the example above declares a character function to have the length specified by the referencing program unit. The length of the value returned is determined by the length declared for function APPLE in the referencing program unit.

How to Reference an External Function

An external function is referenced, or called, when the function name is referenced in an expression. The program unit containing the function reference is known as the referencing program unit. The general form of a function reference is:

name(*a*, ...,*a*)

name

Function name

a

Optional; actual argument that can be one of the following:

A constant (including a symbolic constant or an extended Hollerith constant.)

A variable, array, array section, or array element name

An expression with operators (except a character expression involving concatenation of a dummy argument with length (*)); cannot be an array-valued expression

An intrinsic function name (with a scalar argument)

An external subroutine or function name

A dummy procedure name

Control transfers to a function subprogram when an expression containing the function name is executed in a program unit. Control returns to the calling program unit when a RETURN or END statement is executed in the function.

The function reference can appear anywhere in an expression where an operand of the same type can be used. The actual argument can be a dummy argument that appears in the dummy argument list of the referencing program unit.

The type of the function result is the type of the function name. The arguments must agree in order, number, length (in bytes), and type with the corresponding dummy arguments. An actual argument of type boolean can have a corresponding dummy argument of type integer(8 bytes) or real(8 bytes). An actual argument of type integer or real can have a corresponding dummy argument of type boolean.

An example of a function subprogram is shown in figure 8-1. The example contains the following:

```
      NSUM=NADD(NUMBER)
         :
      END
         :
      FUNCTION NADD(IARRAY)
      INTEGER IARRAY(10)
      N=0
      DO 10 I=1, 10
      N=IARRAY(I) + N
   10 CONTINUE
      NADD=N
      RETURN
      END
```

The function, named NADD, receives an array IARRAY through the argument list. Function NADD sums the values in the array and assigns the result to the function name. In the main program the function is referenced by the statement:

```
NSUM=NADD(NUMBER)
```

The actual argument, NUMBER, is associated with the dummy argument IARRAY. When execution of the function is complete, the result is stored in the variable NSUM and execution continues with the next statement.

Block Data Subprogram

A block data subprogram is a nonexecutable subprogram that specifies initial values for variables and array elements in named common blocks. A program can have more than one block data subprogram.

A block data subprogram is identified by a BLOCK DATA statement. This statement has the form:

BLOCK DATA *name*

name

Name of the block data subprogram. If omitted, name defaults to BLKDAT#

The BLOCK DATA statement must appear as the first statement of the block data subprogram. The name used for the block data subprogram must not be the same as any variables or symbolic constants in the subprogram. The name must not be the same as any other program unit or entry name in the program. Only one block data subprogram can be unnamed.

Block data subprograms can contain IMPLICIT, PARAMETER, DIMENSION, type, COMMON, SAVE, EQUIVALENCE, or DATA statements. A block data subprogram ends with an END statement. Data can be entered into more than one common block in a block data program. All variables having storage in the named common must be specified even if they are not all assigned initial values.

Specifying the name of a block data subprogram in an EXTERNAL statement in the program which makes use of the block data subprogram causes the loader to search the object libraries for the block data subprogram.

Example:

```
BLOCK DATA ANAME
COMMON /CAT/X, Y, Z /DOG/R, S, T
COMPLEX X, Y
DATA X, Y /2*(1.0, 2.7)/, R/7.6543/
END
```

The block data subprogram ANAME enters data into common blocks CAT and DOG. Initial values are defined for variables X and Y in block CAT and variable R in block DOG. No initial values are defined for variables Z, S, or T.

Statement Functions

A statement function consists of a single statement and calculates a single result. It applies only to the program unit containing the statement. The general form of a statement function is:

name(*d, ...,d***) = exp**

name

Function name

d

Dummy argument

exp

Expression

A statement function is a nonexecutable statement. It must appear after the specification statements and before the first executable statement in the program unit.

Examples:

`ADD(A, B, C, D)=A + C + B + D`	ADD is a statement function with dummy arguments A, B, C, and D.
`AVRG(SUM, N)=SUM/N`	AVRG is a statement function with dummy arguments SUM and N.

A statement function name can appear in a type specification statement, but no other specification statements. A statement function can also appear as a common block name in the same program unit.

A statement function name is defined with the value of **exp** after the function executes. During function execution, the actual argument expressions are evaluated, converted if necessary to the types of the corresponding dummy arguments according to the rules for assignment, and passed to the function.

Thus, an actual argument cannot be the same as the name of an array name or a function name. In addition, if a character variable or array element is used as an actual argument, a substring reference to the corresponding dummy argument must not be specified in the statement function expression. The expression of the function is evaluated, and the resulting value is converted, as necessary, to the data type of the function.

A statement function name must not be the same as the name of an array, variable, symbolic constant, intrinsic function, or dummy procedure in the same program unit. The function name cannot be an actual argument and must not appear in an INTRINSIC or EXTERNAL statement. If you use a statement function in a function subprogram, the statement function can contain a reference to the name of the function subprogram or any of its entry names as a variable, but not as a function.

Each variable reference in **exp** can be either a reference to a variable within the same program unit or to a dummy argument of the statement function. Arguments to the function cannot be changed as a result of an external function.

Statement functions within a subprogram can reference dummy arguments that appear in a SUBROUTINE, FUNCTION, or ENTRY statement. Statement function dummy arguments can have the same names as variables defined elsewhere in the same program unit.

How to Reference a Statement Function

A statement function is reference when the name is referenced in an expression. The form of a statement function reference is:

name(*a*, ..., *a*)

name

Statement function name

a

Actual argument that can be one of the following:

A constant (including a symbolic constant or extended Hollerith constant)

A variable, array, array section, or array element name

An expression with operators (except for a character expression involving concatenation of a dummy argument with length (*)); the expression cannot be array-valued

An intrinsic function name (with a scalar argument)

An external subroutine or function name

A dummy subroutine or function name

An alternate return specifier of the form *sl, where sl is the label of an executable statement that appears in the calling program unit

The actual arguments are evaluated and converted to the type of the corresponding dummy arguments; the resulting values are used in place of the corresponding dummy arguments in evaluation of the statement function expression.

A statement function must not directly or indirectly call itself. The statement function reference can appear anywhere in an expression where an operand of the same type can be used.

The type of the statement function result is the type of the statement function name. The arguments must agree in order and number with the corresponding dummy arguments.

A statement function can be referenced only in the program unit where the statement function appears.

Example:

```
F(X, Y)=SQRT(X**2 + Y**2)
    :
Z=F(C(1), C(2))
```

These statements define and reference a statement function. When the function reference is executed, the actual arguments C(1) and C(2) are substituted for the dummy arguments X and Y, and the function is evaluated.

The effect is the same as the following assignment statement:

```
Z=SQRT(C(1)**2 + C(2)**2)
```

Example:

```
AVRG(S, N)=S / N
    :
X=A + B + AVRG(TOT, NSUM)
```

AVRG is a statement function with dummy arguments S and N. When AVRG is referenced, actual arguments TOT and NSUM are substituted for the dummy arguments, the function is evaluated, and the result is used in the expression to calculate a value for X.

How Program Units Pass Values

Program units pass values to one another by

- Argument lists
- Using common blocks

You can use argument lists and common blocks to pass data to external functions and subroutines. You can only use argument lists to pass program unit names to subroutines or external, intrinsic, or statement functions.

Argument Lists

Arguments in the argument list of a SUBROUTINE or FUNCTION statement are called dummy arguments. Arguments in the argument list of a CALL statement or function reference are called actual arguments.

Actual Arguments

The referencing or calling program unit passes actual arguments to the program unit or statement function. The program unit or statement function receives values from the actual arguments and returns values to the referencing or calling program unit.

An actual argument can be one of the following provided that the associated dummy argument is a variable that is not changed during execution of the referenced subprogram or statement function:

- A constant (including a symbolic constant or extended Hollerith constant)
- A variable, array, array section, or array element name
- An expression with operators (except for a character expression involving concatenation of a dummy argument with length (*)); the expression cannot be array-valued
- An intrinsic function name (with a scalar argument)
- An external subroutine or function name
- A dummy subroutine or function name
- An alternate return specifier of the form *sl, where sl is the label of an executable statement that appears in the calling program unit

An actual argument cannot be an array-valued intrinsic or a statement function. An actual argument that is an extended Hollerith constant is treated as if the first element of a one-dimensional boolean array had been specified. However, the associated dummy argument must not be assigned a value during execution of the referenced subprogram or statement function.

An intrinsic function or external function used as an actual argument must be available at the time it is referenced. Also, you must declare the function in an INTRINSIC or EXTERNAL statement (these statements are described in chapter 4, Specification Statements).

An actual argument that is an integer or logical constant expression (with no symbolic constants) is passed as an 8-byte integer or logical value regardless of the actual length, in bytes, of the values in the expression.

Dummy Arguments

Subprograms use dummy arguments to indicate the types of actual arguments, the number of arguments, and whether each argument is a variable, array, subprogram, or statement label. Dummy arguments must conform to the following rules:

- Dummy arguments for statement functions must be variables.
- A dummy argument appearing in a SUBROUTINE, FUNCTION, or ENTRY statement must not appear in an EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC or COMMON (except as a common block name) statement.
- Dummy arguments of type character with a length specification of (*) must not be used as an operand for concatenation, except in a character assignment statement.
- Dummy arguments used in array declarations for adjustable dimensions must be type integer.
- Dummy arguments representing array names must be dimensioned (by a DIMENSION or type statement).

Noncharacter dummy arguments must be equal in length to the associated actual argument.

A dummy argument that is used as if it were a function or subroutine name is called a dummy procedure name. The actual argument associated with a dummy procedure name must be an intrinsic function, external function, subroutine name, or another dummy procedure.

How to Associate Actual and Dummy Arguments

When a subprogram or statement function is referenced, the actual arguments and dummy arguments are matched and each actual argument replaces a corresponding dummy argument. The first (leftmost) actual argument is matched with the first dummy argument, the second actual argument is matched with the second dummy argument, and so forth.

The number of actual arguments and dummy arguments must be the same. Each actual argument must have the same type as its corresponding dummy argument, except that a boolean actual argument can correspond to a dummy argument having any of the arithmetic types. Each integer or logical actual argument must have the same length, in bytes, as its corresponding dummy argument.

If the actual argument is an expression, substring reference, or array element, it is evaluated before the arguments are associated. The subscript or substring value remains constant throughout execution of the subprogram even if variables in the array element reference subscript or substring are changed. If the actual argument is a subprogram name, the subprogram must be available for execution when it is referenced.

A dummy argument is undefined unless it is associated with an actual argument. This can happen if the number of actual arguments in a subprogram reference is less than the number of dummy arguments. It can also happen in subprograms with multiple entry points that do not have the same dummy arguments. (There is no error unless an undefined dummy argument is referenced.)

Argument association can exist at more than one level of subprogram reference and terminates within a program unit when a RETURN or END statement executes.

Example:

```
CALL BSUB(I + J, VAR, X(3))
      ⋮
SUBROUTINE BSUB(N, A, B)
```

The actual arguments are I + J, VAR, and X(3). When the CALL is executed, they become associated with the dummy arguments N, A, and B, respectively.

A subprogram reference can cause a dummy argument to be associated with another dummy argument in the subprogram. Any dummy arguments that become associated with each other can be referenced but must not have values assigned to them during the execution of the subprogram. For example:

```
SUBROUTINE ALPHA(X, Y)
  ⋮
RETURN
END
  ⋮
CALL ALPHA(A, A)
```

The dummy arguments X and Y would each be associated with the actual argument A. X and Y would be associated with each other and therefore must not have values assigned to them.

A subroutine reference can cause a dummy argument to become associated with an entity in a common block. For example:

```
PROGRAM MAIN
COMMON A
CALL ALPHA(A)
  :
SUBROUTINE ALPHA(X)
COMMON Y
```

The actual argument A causes the dummy argument X to become associated with Y, which is in blank common. In this case, X and Y cannot be assigned to during execution of the subroutine.

Variables as Arguments

A variable in a dummy argument list can be associated with a constant, variable, array element, substring, scalar expression, or function reference in the actual argument list. A subprogram can change the value of a scalar dummy argument if the associated actual argument is a variable name, array element name, or scalar substring reference.

A subprogram cannot change the value of a scalar dummy argument if the associated actual argument is a constant, a symbolic constant, a function reference, an expression using operators, or an expression enclosed in parentheses. (An attempt to do so is not diagnosed unless you specify DEBUG_AIDS=PC on the VECTOR_FORTRAN command.)

For character variables, see Character Values as Arguments in this chapter.

Subprogram Names as Arguments

You can pass subroutine names and scalar-valued intrinsic or external function names through the argument list. A dummy argument that corresponds to an actual argument that is a subprogram name must be used as a subprogram name.

If the dummy argument is used as if it were an external function, the corresponding actual argument must be an intrinsic or external function, or dummy procedure name. In this case, the type of the dummy argument must agree with the type of the result of all actual arguments that become associated with the dummy argument.

If the dummy argument has the same name as an intrinsic function but is associated with an actual argument that is an external function, the dummy argument refers to the external function, and not the intrinsic function.

If the dummy argument is referenced as a subroutine, the actual argument must be a subroutine name.

Subprogram names can be passed through more than one level of subprogram reference. At each level, the dummy argument must conform to the preceding rules.

Asterisks as Arguments

A dummy argument that is an asterisk can only appear in the argument list of a SUBROUTINE or ENTRY statement in a subroutine subprogram. The actual argument is an alternate return specifier in the CALL statement.

Arrays as Arguments

An array in a dummy argument list can be associated with an array, array section, or array-valued expression in the actual argument list. A subprogram can change the value of an array dummy argument if the associated actual argument is an array name or array section.

Dummy arguments that represent array names must be dimensioned by a DIMENSION or type statement. The actual argument array and the dummy argument array can differ in the number and size of the dimensions; however, the number and size of dimensions in an actual argument array section or array expression involving operators or parentheses must agree with the number and size of the dimensions in an associated dummy argument declaration.

The following table shows allowable associations between array actual and dummy arguments (a YES indicates that the association is permitted):

	Actual Argument				
Dummy Argument	**Array Name/ Expression**	**Array Section**	**Array Element**	**Assumed-Size Array Name**	**Assumed-Shape Array Name**
Fixed-Size Array	YES [1]	NO	YES	YES	NO
Adjustable Array	YES [1]	NO	YES	YES	NO
Assumed-Size Array	YES	NO	YES	YES	NO
Assumed-Shape Array [2]	YES [1]	YES [1]	NO	NO	YES [1]
Variable	NO	NO	YES	NO	NO

[1] The actual and dummy argument must have the same shape.

[2] Assumed-shape dummy arguments require INTERFACE blocks in the calling program.

For dummy arguments that are assumed-shape arrays, you must use an interface block in the calling program. See the INTERFACE and ENDINTERFACE statements described in chapter 4, Specification Statements.

If the actual argument is a noncharacter array, the size of the dummy argument array must not exceed the size of the actual argument array. Each actual argument array element is associated with the dummy argument array element that has the same subscript value as the actual argument array element.

If the actual argument is an assumed-shape array, each element in the dummy argument array must be in the range d1 through (d1+s1-1) for its dimension; *d1* is the lower dimension bound of the dimension where the subscript is written and *s1* is the size of the corresponding dimension of the actual argument associated with the assumed-shape array.

For example, if a program has the following statements

```
DIMENSION ARRAY_A(5,7)
  :
CALL SUB_B(ARRAY_A)
  :
SUBROUTINE SUB_B(ARRAY_A)
DIMENSION ARRAY_A(1:,0:)
```

then the values of elements in ARRAY_A in the subroutine must be in range

`(1, 1 + 5 - 1)`	for the 1st dimension
`(0, 0 + 7 - 1)`	for the 2nd dimension

Array expressions cannot contain assumed-size arrays.

If the actual argument is a noncharacter array element, the size of the dummy argument cannot exceed ($s + 1 - v$), where s is the size of the actual argument array and v is the subscript value of the array element. For example, if a program has the following statements:

```
DIMENSION ARRAY(20)
      :
CALL CHECK(ARRAY(3))
```

then the value of s is 20, and v is 3. The maximum dummy array size is 18 for the following subroutine:

```
SUBROUTINE CHECK(DUMMY)
DIMENSION DUMMY(18)
      :
```

The actual argument array elements are associated with dummy argument array elements, starting with the first element passed. In the example, DUMMY(2) is associated with ARRAY(4), and DUMMY(18) is associated with ARRAY(20).

An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array or if any variable referenced by the adjustable array declarator is not currently associated with an actual argument or is not in a common block.

If an intrinsic function is referenced with an array name, array section, or array expression as an argument, the function is implicitly referenced n times where n is the size of the array, array section, or array expression. Each implicit reference associates one of the array elements of the array, array section, or array expression actual argument with the corresponding dummy argument. Every element of the array, array section, or array expression is associated.

If there is more than one array, array section, or array expression as an actual argument, each implicit reference associates the same relative element of each array, array section, or array expression. For example:

`INTEGER A(10), B(-5:4)` `B=ATAN2(A, B)`	One of the implicit references to ATAN2 associates A(1) and B(−5) with the corresponding dummy arguments. The next implicit reference would associate A(6) and B(10) with the corresponding dummy arguments.

An array-valued intrinsic function cannot be an actual argument to a subprogram. However, no diagnostic occurs if an array-valued function (from the math library) is an actual argument to a subprogram; the first element in the array is broadcast over all elements of the result array.

Adjustable Dimensions

Adjustable dimensions allow you to create a more general subprogram that can accept varying sizes of array arguments. For example, a subroutine with a fixed array can be declared as:

```
SUBROUTINE SUM(A)
DIMENSION A(10)
```

The maximum array size subroutine SUM can accept is 10 elements. If the same subroutine is to accept an array of any size, it can be written as:

```
SUBROUTINE SUM(A, N)
DIMENSION A(N)
```

In this case, the value N is passed as an actual argument.

Character strings and arrays can also be adjustable, as in the following subroutine:

```
SUBROUTINE MESSAG(X)
CHARACTER X*(*)
PRINT *, X
```

The subroutine declares X with a length of (*) to accept strings of varying size. Note that the length of the string is passed implicitly (rather than explicitly as an actual argument).

Assumed-Size Arrays

Another form of adjustable dimension is the assumed-size array. In this case, the upper bound of the last dimension of the array is specified by an asterisk. The value of the dimension is not passed as an argument. You are responsible for ensuring that the array in the calling program is large enough to contain all the elements stored into it in the subroutine.

Example:

```
    SUBROUTINE CAT(A, M, N, B, C)
    REAL A(M), B(N), C(*)
    DO 10 I=1, M
10  C(I)=A(I)
    DO 20 I=1, N
20  C(I + M)=B(I)
    RETURN
    END
```

Subroutine CAT places the contents of array A, followed by the contents of array B, into array C. The dimension of C in the calling program must be greater than or equal to M + N.

If you use the asterisk form of the adjustable dimension, no subscript checking is done for the array. You should be careful not to reference outside the array bounds.

Assumed-Shape Arrays

Assumed-shape arrays allow you to create a more general subprogram that can accept varying sizes of array arguments. An assumed-shape array is declared when the upper bound of all dimensions is not specified. The value of the dimension is not passed to the subprogram.

You are responsible for ensuring that the array in the calling program is large enough to contain all the elements stored into it in the subprogram.

For example:

`SUBROUTINE B(A)` `DIMENSION A(1: , 1:)`	The variable A can be passed two-dimensional, type real arrays of any size.

Since assumed-shape arrays can be passed contiguous array sections (such as A(1:10:2)), they may not be as efficient as adjustable arrays; but they provide a simple and more reliable call statement or function reference with fewer parameters.

You must use interface blocks in the calling program when calling a subprogram that contains assumed-shape dummy arguments. (See the INTERFACE statement in chapter 4, Specification Statements).

Character Values as Arguments

When character data is passed to a subprogram, both the dummy and actual arguments must be of type character, and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument of type character is greater than the length of the dummy argument, only the leftmost characters of the actual argument, up to the length of the dummy argument, are associated with the dummy argument.

If a dummy argument is an array name or array section, the length restriction applies to the entire array and not to each array element. The length of array elements in the dummy argument can be different from the length of array elements in the actual argument. The total length of the actual argument array must be greater than or equal to the total length of the dummy argument array.

When an actual argument is a character substring, the length of the actual argument is the length of the substring. If the actual argument expression involves concatenation, the sum of the lengths of the operands is the length of the actual argument.

The association for character array elements is basically the same as for noncharacter array elements. The actual argument for a character array can be an array name, array section, array element reference, or character substring name. If the actual argument begins at character position *n* of an array, then the first character position of the dummy argument array becomes associated with character position *n* of the actual argument array, the second character position of the dummy argument array becomes associated with character position *n+1* of the actual argument array, and so forth to the end of the actual argument array.

For example, if a program unit has the following statements:

```
DIMENSION A(2)
CHARACTER A*3
      :
CALL SUB (A)
```

and subroutine SUB has the following statements:

```
SUBROUTINE SUB(B)
DIMENSION B(2)
CHARACTER B*1
```

then the first character of A(1) corresponds to B(1) and the second character of A(1) corresponds to B(2).

Assumed-Length Character Strings

If you specify the length of a type character dummy argument as (*), the dummy argument assumes the length of the associated actual argument for each reference to the subroutine or function. A character dummy argument with length (*) is called an assumed-length character string. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of each array element in the associated actual argument.

Example:

```
PROGRAM MN
CHARACTER*3 CC, A(4)
      :
CALL TSUB(CC, A(1) (2:3))
      :
END
SUBROUTINE TSUB(CHAR, Z)
CHARACTER*(*) CHAR, Z(4)
```

The dummy argument CHAR in subroutine TSUB will have length 3 and each element of the array Z will have length 2.

For Better Performance

Assumed-length character strings require more execution time; replace with constant size strings whenever possible.

Common Blocks

Common blocks can be used to communicate values among program units. The variables and arrays in a common block can be changed in all program units that contain a declaration of that common block. Common blocks are described in chapter 4, Specification Statements.

Example:

```
PROGRAM AVR
COMMON ANUM(10), STORE
      :
SUBROUTINE SUM
COMMON A(10), B
      :
```

The array ANUM in program AVR and the array A in subroutine SUM share the same locations in blank common. The variables STORE and B share the same location. Values stored into ANUM in the main program are available to SUM.

Example:

```
PROGRAM COM
COMMON / CB / ARR(3)
      :
SUBROUTINE SUB
COMMON / CB / A, B, C
```

Variable A in SUB shares the same location as ARR(1) in the main program, B shares the same location as ARR(2), and C shares the same location as ARR(3).

A reference to data in a common block is valid if the data is defined and is the same type as the type of the name used in the main program or subprogram. The exceptions to agreement between the type in common and the type of the reference are:

- Either part of a complex entity can be referenced as real.
- A boolean entity can be referenced as an 8-byte integer.

In a subprogram, entities declared in a named or blank common block remain defined at all times and do not become undefined on returning from a subprogram.

ENTRY and RETURN Statements

Each subprogram has a primary entry point established by the SUBROUTINE or FUNCTION statement that begins the program unit. A subroutine call or function reference usually invokes the subprogram at the primary entry point, and the first statement executed is the first executable statement in the program unit. You can use ENTRY statements to define other entry points.

The RETURN statement returns control from a referenced subprogram back to the calling or referencing program unit.

ENTRY statement

The ENTRY statement defines an entry point for a subroutine or function subprogram. This statement has the form:

ENTRY epname(*d*, ..., *d***)**

epname

Entry point name

d

Dummy argument that can be one of the following:

A variable name

An array name

An array section reference

A dummy procedure name

An asterisk, in a subroutine only

An ENTRY statement can appear anywhere in a subprogram after the SUBROUTINE or FUNCTION statement except:

- Between a block IF statement and its corresponding END IF statement
- Between a block WHERE statement and its corresponding ENDWHERE statement
- Between a DO statement and the terminating statement of the DO loop.

If the entry point name in a function subprogram is of type character, each entry point name and the function subprogram name must be type character and of the same length.

When an entry name is used to reference a subprogram, execution begins with the first executable statement that follows the referenced entry point. An entry name is available for reference in any program unit, except in the subprogram that contains the entry name. The entry name can appear in an EXTERNAL statement and (for a function entry name) in a type statement.

Each reference to a subprogram must use an actual argument list that corresponds in number of arguments and type of arguments to the dummy argument list in the corresponding ENTRY statement. The dummy arguments for an entry point can therefore be different from the dummy arguments for the primary entry point or another entry point. Type agreement is not required for actual arguments that have no type, such as a dummy subroutine name.

If an entry point name is of type 2-, 4-, or 8-byte integer or logical, all other entry point names in the same function or subroutine must be of the same length. Similarly, if an entry point name is of type byte, all other entry point names in the same function or subprogram must be of type byte.

A dummy argument cannot be used in an executable statement of a subprogram unless the argument has appeared in a physically preceding FUNCTION, SUBROUTINE, or ENTRY statement.

Example:

```
SUBROUTINE ALPHA(ARR, N)
DIMENSION ARR(N)
DO 20 I=1, N
      :
ENTRY BETA(ARR, N, K)
DO 40 I=1, N, K
      :
```

Subroutine ALPHA can be entered either through entry point ALPHA, in which case the first statement executed is DO 20 I=1, N, or through entry point BETA, in which case the first statement executed is DO 40 I=1, N, K. Note that the SUBROUTINE and ENTRY argument lists legally contain different numbers of arguments. If the subroutine is entered through ALPHA, K cannot be referenced.

RETURN Statement

The RETURN statement transfers control from a referenced subprogram back to the referencing program unit. This statement has the form:

RETURN *exp*

exp

Scalar arithmetic or boolean expression which can only be used when returning from a subroutine. If *exp* is not of type integer, the value INT(exp) is used.

The value of *exp* determines the statement to which control returns. If the value of exp is 1, control returns to the first statement label specified in the actual argument list; if the value of *exp* is 2, control returns to the second label, and so forth. If *exp* is less than 1 or greater than the number of labels in the actual argument list, control returns to the statement following the CALL statement.

When a RETURN statement without the optional expression is executed, control transfers to the statement immediately following the statement that called the referenced subprogram. If the referenced subprogram is a function subprogram, the function value is returned through the function name, the expression containing the function reference is evaluated, the result is stored, and execution continues.

RETURN statements are valid in main programs and in subroutine and function subprograms. Execution of a RETURN statement in a main program has the same effect as a STOP or END statement: the program terminates and control returns to the operating system.

A subprogram can contain more than one RETURN statement. You can place them anywhere in the subprogram where a return operation is desired. Note that if a RETURN statement is not the last executable statement of a subprogram, the statement following the RETURN can be executed only as a result of a flow control statement, such as IF or GO TO, that transfers control to that statement. You can omit the RETURN statement entirely, in which case control returns to the calling program unit when an END statement is encountered.

If *exp* is specified, the expression corresponds to a statement label in the calling program unit. The statement labels eligible must be included in the actual argument list, with each label preceded by an asterisk. In the SUBROUTINE statement, the dummy argument corresponding to each statement label actual argument must be an asterisk.

Intrinsic Functions 9

Generic and Specific Names ... 9-3
Elemental and Array-Processing Intrinsic Functions ... 9-4
Mathematical Intrinsic Functions ... 9-6

Function Descriptions ... 9-8
ABS ... 9-8
ACOS ... 9-8
AIMAG ... 9-9
AINT ... 9-9
ALL ... 9-10
ALLOCATED ... 9-10
ALOG ... 9-11
ALOG10 ... 9-11
AMAX0 ... 9-11
AMAX1 ... 9-12
AMIN0 ... 9-12
AMIN1 ... 9-12
AMOD ... 9-13
AND ... 9-13
ANINT ... 9-14
ANY ... 9-14
ASIN ... 9-15
ATAN ... 9-15
ATANH ... 9-16
ATAN2 ... 9-16
BOOL ... 9-17
CABS ... 9-17
CCOS ... 9-18
CEXP ... 9-18
CHAR ... 9-19
CLOG ... 9-19
CMPLX ... 9-20
COMPL ... 9-20
CONJG ... 9-21
COS ... 9-21
COSD ... 9-22
COSH ... 9-22
COTAN ... 9-23
COUNT ... 9-23
CSIN ... 9-24
CSQRT ... 9-24
DABS ... 9-24
DACOS ... 9-25
DASIN ... 9-25
DATAN ... 9-25
DATAN2 ... 9-26
DBLE ... 9-26
DCOS ... 9-27
DCOSH ... 9-27
DDIM ... 9-27
DEXP ... 9-28
DIM ... 9-28
DINT ... 9-29

DLOG 9-29
DLOG10 9-29
DMAX1 9-30
DMIN1 9-30
DMOD 9-30
DNINT 9-31
DOTPRODUCT 9-31
DPROD 9-32
DSIGN 9-32
DSIN 9-32
DSINH 9-33
DSQRT 9-33
DTAN 9-33
DTANH 9-34
EQV 9-34
ERF 9-35
ERFC 9-35
EXP 9-36
EXTB 9-36
FLOAT 9-37
IABS 9-37
ICHAR 9-37
IDIM 9-38
IDINT 9-38
IDNINT 9-38
IFIX 9-39
INDEX 9-39
INSB 9-40
INT 9-41
ISIGN 9-41
LBOUND 9-42
LEN 9-42
LGE 9-43
LGT 9-43
LLE 9-44
LLT 9-44
LOG 9-45
LOG10 9-45
MASK 9-46
MATMUL 9-47
MAX 9-48
MAXVAL 9-48
MAX0 9-49
MAX1 9-49
MERGE 9-50
MIN 9-50
MINVAL 9-51
MIN0 9-51
MIN1 9-52
MOD 9-52
NEQV 9-53
NINT 9-53
OR 9-54
PACK 9-54
PRODUCT 9-55
PTR 9-56
RANF 9-56

RANK 9-57
REAL 9-57
SEQ 9-58
SHAPE 9-59
SHIFT 9-60
SIGN 9-60
SIN 9-61
SIND 9-61
SINH 9-62
SIZE 9-62
SNGL 9-63
SQRT 9-63
SUM 9-64
SUM1S 9-65
TAN 9-65
TAND 9-66
TANH 9-66
UBOUND 9-67
UNPACK 9-68
XOR 9-69

Intrinsic Functions 9

An intrinsic function is a pre-defined function that returns a single or array value. A program references intrinsic functions the same way it references external (user-defined) functions.

If, in a particular program unit, a variable, array, or statement function is declared with the same name as an intrinsic function, the intrinsic function of the same name cannot be referenced in that program unit. If an external function name is the same as an intrinsic function, use of the name references the intrinsic function, unless you declare the name as an external function with the EXTERNAL statement described in chapter 4, Specification Statements.

Intrinsic functions are typed by default and need not appear in any type or IMPLICIT statement in the program. (Explicitly typing a generic intrinsic function name does not remove the generic properties of the name.)

There are two ways to compile intrinsic functions:

- Inline, from the FORTRAN Version 2 compiler
- From an external library of mathematical functions called the Math Library

Some examples of functions compiled inline are ABS and SIGN, and some examples of Math Library functions are ATANH and CEXP.

The following intrinsic functions are always compiled inline unless you specify EXPRESSION_EVALUATION= REFERENCE or OPTIMIZATION_LEVEL=DEBUG on the VECTOR_FORTRAN command (then the Math Library version is used):

- ACOS
- ALOG
- ALOG10
- ASIN
- ATAN
- COS
- EXP
- SIN
- SQRT
- TAN

Although all methods generate the same results, intrinsic functions that are compiled inline usually execute faster since no calls are made to the Math Library. Functions from the Math Library; however, generate more precise error messages should an execution error occur. This is helpful if the SYSTEM or SYSTEMC subroutines are used in error processing.

Routines that are in the Math Library are normally accessed through the call-by-value calling procedure. To access an intrinsic function through the call-by-reference calling procedure, specify EXPRESSION_EVALUATION=REFERENCE on the VECTOR_FORTRAN command. The Math Library also contains vector versions of some intrinsic functions. Vector versions of intrinsic functions are used when a program is compiled with VECTORIZATION_LEVEL=HIGH.

The Math Library for NOS/VE manuals describes the mathematical algorithms and entry points for the intrinsic functions described in this chapter.

NOTE

You should ensure that real, double precision, and complex arguments to intrinsic functions are in normalized standard floating point form, as unnormalized or nonstandard arguments can cause undefined results. FORTRAN automatically normalizes all real, double precision, and complex constants, and results of all floating point operations with standard normalized or zero operands are normalized or zero. However, it is possible to generate unnormalized or non-standard operands by means of boolean expressions, equivalencing, or various input operations.

Generic and Specific Names

Certain intrinsic functions have a generic name and one or more specific names. For these functions, either the generic name or one of the specific names can be used. The generic name provides more flexibility because it can be used with any of the valid data types; except for functions performing type conversion, nearest integer, and absolute value with a complex argument, the type of the argument determines the type of the result.

Integer arguments to the functions can be of any length (or of type byte) unless otherwise noted in the function description. Similarly, logical arguments to the functions can be of any length unless otherwise noted in the function description. Real arguments must be 8 bytes in length; 16-byte real arguments are treated as double precision and are only allowed if a function accepts type double precision arguments.

A function accepting integer, byte, real, complex or double precision type arguments also accepts boolean arguments. A boolean argument is converted to integer, if integer is an allowable argument type; otherwise, it is converted to real, if real is an allowable argument type; otherwise, it is converted to double precision or complex, before computation.

Only a specific name can be used as an actual argument when passing the function name to an external function subroutine. Using a specific name requires a specific argument type. For example, the generic function name LOG computes the natural logarithm of an argument. Its argument can be real, double precision, complex or boolean (converted to real). The type of the result is the same as the type of the argument (real if the argument was boolean).

Specific function names ALOG, DLOG, and CLOG also compute the natural logarithm. The specific function name ALOG computes the log of a real or boolean argument and returns the result. Likewise, the specific name DLOG is for double precision (or boolean) arguments and double precision results and the specific name CLOG is for complex (or boolean) arguments and complex results.

For example:

`AVAL=LOG(B)`	B (and AVAL) can be of boolean, real, or complex type.
`AVAL=ALOG(B)`	B must be of type real.

Elemental and Array-Processing Intrinsic Functions

An intrinsic function is elemental if the result can be determined on an element-by-element basis when applied to an array-valued argument. An array-valued argument is an array, array section, or array expression. All of the elemental intrinsic functions accept array-valued arguments.

If an elemental intrinsic function is referenced with an array-valued argument, the function is implicitly referenced *n* times where *n* is the size of the array, array section, or array expression. See Arrays as Arguments in chapter 8, Program Units.

An intrinsic function is an array-processing function when the result cannot be determined on an element-by-element basis. The array-processing intrinsic functions are:

ALL

ALLOCATED

ANY

COUNT

DOTPRODUCT

LBOUND

MATMUL

MAXVAL

MINVAL

PACK

PRODUCT

RANK

SEQ

SHAPE

SIZE

SUM

UBOUND

UNPACK

You cannot pass array-processing intrinsic functions as actual arguments to a subprogram or statement function.

NOTE

The results of certain array-processing functions (such as SUM and MATMUL), when used with arrays containing elements whose magnitudes vary widely, may differ slightly from the results of the equivalent operations performed using scalar statements. This occurs because the order of the operations performed by the functions may differ from that of the scalar statements.

Mathematical Intrinsic Functions

Table 9-1 shows the domain and range for a subset of the mathematical intrinsic functions.

Table 9-1. Mathematical Intrisic Functions

Function	Domain	Range
ACOS(a) DACOS(a)	$\lvert a \rvert \le 1$	$0 \le \mathrm{ACOS(a)} \le$ pi
ASIN(a) DASIN(a)	$\lvert a \rvert \le 1$	$-\mathrm{pi}/2 \le \mathrm{ASIN(a)} \le \mathrm{pi}/2$
ATAN(a) DATAN(a)	$-\mathrm{infinity} \le a \le \mathrm{infinity}$	$-\mathrm{pi}/2 \le \mathrm{ATAN(a)} \le \mathrm{pi}/2$
ATAN2(a1,a2) DATAN2(a1,a2)	$a2<0,\ a1<0$ $a2<0,\ a1 \ge 0$ $a2=0,\ a1<0$ $a2=0,\ a1>0$ $a2>0,\ a1<0$ $a2>0,\ a1 \ge 0$ $a2=0,\ a1=0$ (error)	$-\mathrm{pi} < \mathrm{ATAN2(a1,a2)} < -\mathrm{pi}/2$ $\mathrm{pi}/2 \le \mathrm{ATAN2(a1,a2)} \le \mathrm{pi}$ $\mathrm{ATAN2(a1,a2)} = -\mathrm{pi}/2$ $\mathrm{ATAN2(a1,a2)} = \mathrm{pi}/2$ $-\mathrm{pi}/2 < \mathrm{ATAN2(a1,a2)} < 0$ $0 \le \mathrm{ATAN2(a1,a2)} < \mathrm{pi}/2$
ATANH(a)	$\lvert a \rvert \le 1$	
COS(a) DCOS(a)	$\lvert a \rvert < 2^{**}47$	$-1 \le \mathrm{COS(a)} \le 1$
COTAN(a)	$\lvert a \rvert < 2^{**}47$	
CCOS(ar,ai)	$\lvert ar \rvert < 2^{**}47$ $\lvert ai \rvert < 4095^{*}\log(2)$	$-1 \le \mathrm{CCOS(x)} \le 1$ where $x=(ar)+(ai)i$
COSD(a)	$\lvert a \rvert < 2^{**}47$	$-1 \le \mathrm{COSD(a)} \le 1$
COSH(a) DCOSH(a)	$\lvert a \rvert < 4095^{*}\log(2)$	$\mathrm{COSH(a)} \ge 1$ $\mathrm{DCOSH(a)} \ge 1$
ERF(a)	$-\mathrm{infinity} \le a \le \mathrm{infinity}$	$-1 \le \mathrm{ERF(a)} \le 1$
ERFC(a)	$-\mathrm{infinity} \le a \le 25.923$	$0 \le \mathrm{ERFC(a)} \le 2$
EXP(x) DEXP(x)	$x < 4095^{*}\mathrm{LOG}(2)$ $x \ge -4097^{*}\mathrm{LOG}(2)$	

(Continued)

Table 9-1. Mathematical Intrisic Functions *(Continued)*

Function	Domain	Range
CEXP(ar,ai)	ar<4095*LOG(2) ar≥-4097*LOG(2) |ai|<2**47	
LOG(a) ALOG(a) DLOG(a)	a>0	|LOG(a)|<4095*LOG(z)
CLOG(ar,ai)	(ar, ai) ≠ (0,0) (ar**2+ai**2)**1/2 in machine range	- pi<CLOG(ai)<pi
LOG10(a) ALOG10(a) DLOG10(a)	a>0	|LOG10(a)|<4095*LOG(2) base 10
SIN(a) DSIN(a)	|a|<2**47	-1≤SIN(a)≤1
CSIN(ar,ai)	|ar|<2**47 |ai|<4095*log(2)	
SIND(a)	|a|<2**47	-1≤SIND(a)≤1
SINH(a) DSINH(a)	|a|<4095*log(2)	
SQRT(a) DSQRT(a)	a≥0	SQRT(a)≥0
CSQRT(ar,ai)	(ar**2+ai**2)**1/2+|ar| in machine range	value in right half of plane (ar≥0)
TAN(a) DTAN(a)	|a|<2**47	
TAND(a)	|a|<2**47 a cannot be exact odd multiple of 90	
TANH(a)		-1≤TANH(a)≤1

The following pages list the intrinsic functions in alphabetical order of generic name or, where no generic name exists, of specific name.

Function Descriptions

Following are descriptions of the intrinsic functions. The generic and specific names are listed in alphabetical order.

ABS

Purpose	Returns the absolute value (magnitude).
Format	**ABS(a)**
Function	Generic
Argument type	Boolean, byte, complex, double precision, integer, or real
Result type	Same as the argument, except boolean is converted to integer and complex is converted to real.
Specific names	IABS For byte or integer arguments only ABS For real arguments only DABS For double precision arguments only CABS For complex arguments only
Remarks	For integer arguments, the result has the same length, in bytes, as the argument. For a complex argument, the result is the square root of (ar**2 + ai**2), where ar is the real part of the argument and ai is the imaginary part.

ACOS

Purpose	Returns the arcosine.
Format	**ACOS(a)**
Function	Generic
Argument type	Boolean, double precision, or real
Result type	Same as the argument, except boolean is converted to real.
Specific names	ACOS For real arguments only DACOS For double precision arguments only
Generic name	None
Remarks	The result is in radians.

AIMAG

Purpose	Returns an imaginary part.
Format	**AIMAG(a)**
Function	Specific
Argument type	Boolean or complex
Result type	Real
Generic name	None
Remarks	The result is ai, where the complex argument is (ar, ai).

AINT

Purpose	Returns a whole number after truncation.
Format	**AINT(a)**
Function	Generic
Argument type	Boolean, double precision, or real
Result type	Same as the argument except boolean is converted to real.
Remarks	If $\|a\|<1$, the result is 0. If $\|a\|>1$, the result is the largest whole number with the same sign as the argument a that does not exceed $\|a\|$.
Specific names	AINT For real arguments only DINT For double precision arguments only

Control Data Extension

ALL

Purpose	Returns the value true if every element of a1, along the optional dimension specification a2, has the logical value true.
Format	**ALL(a1,*a2*)**
Function	Generic
Argument type	a1: logical a2: boolean, byte, or integer
Result type	Logical
Remarks	Argument a1 must be an array, array section, or array expression. If you omit a2, the function is applied to all elements of a1 to return a scalar value. The result has the same length, in bytes, as argument a1. If a1 has size zero, then the result has size zero unless dimension a2 of a1 is the only dimension of size zero; in which case, the result has a nonzero size and all elements have the same value.

ALLOCATED

Purpose	Returns a scalar logical value indicating whether or not an allocatable array is allocated.
Format	**ALLOCATED(a1)**
Function	Generic
Argument type	Any
Result type	8-byte logical
Remarks	Argument a1 must be an allocatable array name.

End of Control Data Extension

ALOG

Purpose	Returns the natural logarithm (logarithm base 10).
Format	**ALOG(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	LOG
Remarks	The argument must be greater than zero.

ALOG10

Purpose	Returns the common logarithm (logarithm base e).
Format	**ALOG10(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	LOG10
Remarks	The argument must be greater than zero.

AMAX0

Purpose	Returns the largest value from 2 through 500 arguments.
Format	**AMAXO(a,...,*a*)**
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Real

AMAX1

Purpose	Returns the largest value from 2 through 500 arguments.
Format	**AMAX1(a,** ..., *a*)
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	MAX

AMIN0

Purpose	Returns the smallest value from 2 through 500 arguments.
Format	**AMINO(a,** ..., *a*)
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Real
Generic name	None

AMIN1

Purpose	Returns the smallest value from 2 through 500 arguments.
Format	**AMIN1(a,** ...,*a*)
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	MIN

AMOD

Purpose	Returns the remainder of a1 divided by a2 (a1 modulus a2). The result is a1-INT(a1/a2)*a2.
Format	**AMOD(a1,a2)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	MOD
Remarks	If a2 is zero, results are undefined.
	The result is a1-INT(a1/a2)*a2.

Control Data Extension

AND

Purpose	Returns the boolean product of 2 through 500 arguments.
Format	**AND(a,** ..., *a*)
Function	Specific
Argument type	Any type but character
Result type	Boolean
Generic name	None
Remarks	The result is the same as for the boolean. AND. operator.

End of Control Data Extension

ANINT

Purpose	Returns the nearest whole number.
Format	**ANINT(a)**
Function	Generic
Argument type	Boolean, double precision, or real
Result type	Same as the argument except boolean is converted to real.
Specific names	ANINT For real arguments only DNINT For double precision arguments only
Remarks	The result is defined as INT(a+.5) if a is positive or zero, and INT(a-.5) if **a** is negative.

Control Data Extension

ANY

Purpose	Returns the logical value true if one or more elements of a1, along the optional dimension specification a2, has the logical value true.
Format	**ANY(a1,** *a2***)**
Function	Generic
Argument type	a1: logical a2: boolean, byte or integer
Result type	Logical
Remarks	Argument a1 must be an array, array section, or array expression. If you omit a2, the function is applied to all elements of a1 to return a scalar value. The result has the same length, in bytes, as argument a1. If a1 has size zero, then the result has size zero unless a2 of a1 is the only dimension of size zero. In this case, the result has a nonzero size and all elements have the same value.

End of Control Data Extension

ASIN

Purpose	Returns the arcsine.
Format	**ASIN(a)**
Function	Generic
Argument type	Boolean, double precision, or real
Result type	Same as the argument except a boolean argument is converted to real.
Specific names	ASIN For real arguments only DASIN For double precision arguments only
Remarks	The result is in radians.

ATAN

Purpose	Returns the arctangent.
Format	**ATAN(a)**
Function	Generic
Argument type	Boolean, double precision, or real
Result type	Same as the argument except boolean is converted to real.
Specific names	ATAN For real arguments only DATAN For double precision arguments only
Remarks	The result is in radians.

Control Data Extension

ATANH

Purpose	Returns the hyperbolic arctangent.
Format	**ATANH(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	None

End of Control Data Extension

ATAN2

Purpose	Returns the arctangent of a1/a2.
Format	**ATAN2(a1, a2)**
Function	Generic
Argument type	Boolean, double precision, or real
Result type	Same as the argument except boolean is converted to real.
Specific names	ATAN2 For real arguments only DATAN2 For double precision arguments only
Remarks	The result is as follows:

Arguments	Result
a2<0, a1<0	-pi+arctan(a1/a2)
a2=0, a1<0	-pi/2
a2=0, a1>0	pi/2
a2=0, a1=0	error
a2<0, a1≥0	pi+arctan(a1/a2)
a2>0	arctan(a1/a2)

The arguments must both not be zero.

The result is in radians.

Control Data Extension

BOOL

Purpose: Performs type conversion and returns a boolean value.

Format: **BOOL(a)**

Function: Generic

Argument type: Any type but logical

Result type: Boolean

Remarks: For an integer, real, or boolean argument, the result is the bit string constituting the data.

For a double precision or complex argument, the result is the bit string after conversion of the argument to real with REAL(a).

For a character argument, the result is the value of the boolean string constant nf, where n is the length and f is the character value; if n is greater than 8, the rightmost characters are truncated.

Byte and 2- or 4-byte integer arguments are sign-extended to 8 bytes before being passed to BOOL.

End of Control Data Extension

CABS

Purpose: Returns the absolute value.

Format: **CABS(a)**

Function: Specific

Argument type: Boolean or complex

Result type: Real

Generic Name: ABS

Remarks: The result is the square root of (ar**2+ai**2), where ar is the real part of the argument and ai is the imaginary part.

CCOS

Purpose	Returns the cosine.
Format	**CCOS(a)**
Function	Specific
Argument type	Boolean or complex
Result type	Complex
Generic Name	COS

CEXP

Purpose	Returns the value of e raised to a complex power.
Format	**CEXP(a)**
Function	Specific
Argument type	Boolean or complex
Result type	Complex
Generic Name	EXP

CHAR

Purpose	Returns the character value in the ith position of a collating sequence.
Format	**CHAR(i)**
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Character (length one)
Generic name	None
Remarks	The value returned depends on the collating sequence in use. If the ASCII collating sequence is used, the argument must be in the range 0 through 255; the first character in the collating sequence corresponds to value 0, the second character to value 1, the third to value 2, and so forth. The result is the selection of a single character from the collating sequence. (If you specify an argument greater than 255 and compile with DEFAULT_COLLATION=FIXED, MOD(i, 256) is used. If you specify an argument greater than 255 and compile with DEFAULT_COLLATION=USER, a runtime error issue is issued.) If, in a user-specified collating sequence, more than one character has weight i, any of the characters can be returned. User-specified collating sequences are explained in chapter 10.

CLOG

Purpose	Returns the natural logarithm (logarithm base e).
Format	**CLOG(a)**
Function	Specific
Argument type	Boolean or complex
Result type	Complex
Generic Name	LOG

CMPLX

Purpose	Performs type conversion and returns a complex value.
Format	**CMPLX(a1**,*a2*)
Function	Generic
Argument type	Boolean, byte, complex, double precision, integer, or real
Result type	Complex
Remarks	A boolean argument is treated as a bit string and is not changed.

If both arguments are specified, they cannot be complex.

For two arguments a1 and a2, the arguments must be of the same type (one or both can be of type boolean).

REAL(a1) used as the real part and REAL(a2) used as the imaginary part.

For a single argument, the result is complex, with REAL(a) used as the real part and the imaginary part set to zero.

For a single complex argument, the result is the same as the argument.

Control Data Extension

COMPL

Purpose	Returns a complemented value.
Format	**COMPL(a)**
Function	Specific
Argument type	Any type except character
Result type	Boolean
Generic name	None
Remarks	The result is the value of the logical operator .NOT. on a boolean value.

If the argument is not boolean, the argument is converted with BOOL(a).

End of Control Data Extension

CONJG

Purpose	Returns the conjugate.
Format	**CONJG(a)**
Function	Specific
Argument type	Boolean or complex
Generic name	None
Result type	Complex
Remark	For a complex argument (ar,ai), the result is (ar,-ai), with the imaginary part negated.

COS

Purpose	Returns the cosine.
Format	**COS(a)**
Function	Generic
Argument type	Boolean, complex, double precision, or real
Result type	Same as the argument except boolean is converted to real.
Specific names	COS For real arguments only CCOS For complex arguments only DCOS For double precision arguments only
Remark	The argument is in radians.

Control Data Extension

COSD

Purpose	Returns the cosine.
Format	**COSD(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	None
Remark	The argument is in degrees.

End of Control Data Extension

COSH

Purpose	Returns the hyperbolic cosine.
Format	**COSH(a)**
Function	Generic
Argument type	Boolean, double precision, or real
Result type	Same as the argument except boolean is converted to real.
Specific names	COSH For real arguments only DCOSH For double precision arguments only

Control Data Extension

COTAN

Purpose	Returns the cotangent.
Format	**COTAN(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	None
Remarks	COTAN first reduces the argument a by modulo 2*pi. The argument is in radians.

COUNT

Purpose	Returns the number of true elements in a1 along the optional dimension specification a2.
Format	**COUNT(a1,*a2*)**
Function	Specific
Argument type	a1: logical a2: boolean, byte, or integer
Result type	8-byte integer
Remarks	Argument a1 must be an array, array section, or array expression. If a2 is omitted, the function is applied to all elements of a1 to yield a scalar value. If a1 has size zero, then the result has size zero unless dimension a2 of a1 is the only dimension of size zero; in which case, the result has a nonzero size and all elements have the same value.

End of Control Data Extension

CSIN

Purpose	Returns the sine.
Format	**CSIN(a)**
Function	Specific
Argument type	Boolean or complex
Result type	Complex
Generic Name	SIN

CSQRT

Purpose	Returns the square root.
Format	**CSQRT(a)**
Function	Specific
Argument type	Boolean or complex
Result type	Complex
Generic Name	SQRT

DABS

Purpose	Returns the absolute value (magnitude).
Format	**DABS(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	ABS

DACOS

Purpose	Returns the arccosine.
Format	**DACOS(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	ACOS

DASIN

Purpose	Returns the arcsine.
Format	**DASIN(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	ASIN

DATAN

Purpose	Returns the arctangent.
Format	**DATAN(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	ATAN

DATAN2

Purpose

Returns the arctangent of a1/a2.

Format

DATAN2(a1, a2)

Function

Specific

Argument type

Boolean or double precision

Result type

Double precision

Generic name

ATAN2

Remarks

The result is as follows:

Arguments	Result
a2<0, a1<0	-pi+arctan(a1/a2)
a2=0, a1<0	-pi/2
a2=0, a1>0	pi/2
a2=0, a1=0	error
a2<0, a1≥0	pi+arctan(a1/a2)
a2>0	arctan(a1/a2)

The result is in radians.

DBLE

Purpose

Converts the argument to double precision.

Format

DBLE(a)

Function

Generic

Argument type

Boolean, byte, integer, real, double precision, or complex

Result type

Double precision

Remarks

A boolean argument is treated as a bit string and is not changed.

For an integer, byte, or real argument, the result has as much precision of the significant part of the argument as the double precision field can contain.

For a double precision argument, the result is the argument.

For a complex argument, the real part is used, and the result has as much precision of the significant part of the real part of the argument as the double precision field can contain.

DCOS

Purpose	Returns the cosine.
Format	**DCOS(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	COS

DCOSH

Purpose	Returns the hyperbolic cosine.
Format	**DCOSH(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	COSH

DDIM

Purpose	Returns a positive difference.
Format	**DDIM(a1, a2)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	DIM
Remarks	The result is the value of a1-a2 if a1 is greater than or equal to a2; if a1 is less than a2, it returns zero.

DEXP

Purpose	Returns the exponential result.
Format	**DEXP(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	EXP

DIM

Purpose	Returns a positive difference.
Format	**DIM(a1, a2)**
Function	Generic
Argument type	Boolean, byte, integer, real, or double precision
Result type	Same as the argument except boolean is converted to real.
Specific names	IDIM For integer or byte arguments only DIM For real arguments only DDIM For double precision arguments only
Remarks	The result is a1-a2 if a1 is greater than or equal to a2. The result is zero if a1 is less than a2. Both arguments must be the same type unless one is of type boolean (converted to real).

DINT

Purpose	Returns a whole number after truncation.
Format	**DINT(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	AINT

DLOG

Purpose	Returns the natural logarithm (base e).
Format	**DLOG(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	LOG

DLOG10

Purpose	Returns the common logarithm (logarithm base 10).
Format	**DLOG10(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	LOG10

DMAX1

Purpose	Returns the largest value from 2 through 500 arguments.
Format	**DMAX1(a,**...,*a*)
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	MAX

DMIN1

Purpose	Returns the smallest value of 2 through 500 arguments.
Format	**DMIN1(a,**...,*a*)
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	MIN

DMOD

Purpose	Returns a1 modulus a2 (the remainder of a1 divided by a2).
Format	**DMOD(a1, a2)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	MOD
Remarks	The result is a1-(INT(a1/a2)*a2).
	If a2 is zero, results are undefined.

DNINT

Purpose	Returns the nearest whole number.
Format	**DNINT(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	ANINT
Remarks	The result is defined as INT(a+.5) if a is positive or zero, and INT(a-.5) if a is negative.

Control Data Extension

DOTPRODUCT

Purpose	Returns the dot product of a1 and a2.
Format	**DOTPRODUCT(a1,*a2*)**
Function	Generic
Argument type	a1,a2: Boolean, integer, byte, real, double precision, or complex
Result type	Same type as the argument except boolean is converted to integer.
Remarks	The dot product of two arrays is the sum of the products of corresponding elements. The result is a scalar value. Integer results are the same length, in bytes, as the argument. Integer arguments must have the same length. Both arguments must have the same shape. Both arguments must be one-dimensional array expressions.

End of Control Data Extension

DPROD

Purpose	Returns the double precision product of two boolean or real arguments.
Format	**DPROD(a1, a2)**
Function	Specific
Argument type	Boolean or real
Result type	Double precision
Generic name	None
Remarks	The result is defined as a1*a2.

DSIGN

Purpose	Performs a transfer of sign.
Format	**DSIGN(a1, a2)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	SIGN
Remarks	The result is defined as |a1| if a2 is positive or zero, and as -|a1| if a2 is negative.

DSIN

Purpose	Returns the sine.
Format	**DSIN(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	SIN
Remarks	The argument is in radians.

DSINH

Purpose	Returns the hyperbolic sine.
Format	**DSINH(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	SINH

DSQRT

Purpose	Returns the square root.
Format	**DSQRT(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	SQRT
Remarks	The argument must not be negative.

DTAN

Purpose	Returns the tangent.
Format	**DTAN(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Double precision
Generic Name	TAN
Remarks	The argument is in radians.

DTANH

Purpose Returns the hyperbolic tangent.

Format **DTANH(a)**

Function Specific

Argument type Boolean or double precision

Result type Double precision

Generic Name TANH

Control Data Extension

EQV

Purpose Returns the equivalence of 2 through 500 arguments.

Format **EQV(a,...,*a*)**

Function Specific

Argument type Any type except character

Result type Boolean

Generic Name None

Remarks The result is the same as for the boolean .EQV. operator.

End of Control Data Extension

Control Data Extension

ERF

Purpose	Returns an error function result.
Format	**ERF(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic Name	None
Remarks	The mathematical definition is as follows:

$$\mathrm{ERF}(x) = 2/\sqrt{pi} \int_{0}^{x} e^{-t^2}\,dt$$

ERFC

Purpose	Returns a complementary error function result.
Format	**ERFC(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic Name	None
Remarks	The result is 1-ERF(a). The mathematical definition of ERFC is a follows:

$$\mathrm{ERFC}(x) = 2/\sqrt{pi} \int_{x}^{\infty} e^{-t^2}\,dt$$

End of Control Data Extension

EXP

Purpose	Returns an exponential result (e**a).
Format	**EXP(a)**
Function	Generic
Argument type	Boolean, real, double precision, or complex
Result type	Same as the argument except boolean is converted to real.
Specific names	EXP For real arguments only DEXP For double precision arguments only CEXP For complex arguments only

Control Data Extension

EXTB

Purpose	Returns extracted bits from a1; a2 specifies the ordinal of the first bit to be extracted and a3 specifies the number of bits to be extracted.
Format	**EXTB(a1, a2, a3)**
Function	Generic
Argument type	a1: any type except character a2, a3: boolean, integer, or byte
Result type	Boolean
Remarks	The result is undefined if a2 is less than zero or greater than 63 or if a3 is less than zero, or if (a2+a3) is greater than 64. Bits are numbered from the left starting with zero. The argument is converted to boolean using BOOL(a1). Integer arguments must be of the same length.

End of Control Data Extension

FLOAT

Purpose	Returns the value after the conversion to real.
Format	**FLOAT(a)**
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Same as the argument except boolean is converted to real.
Generic Name	REAL

IABS

Purpose	Returns the absolute value (magnitude).
Format	**IABS(a)**
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Same as the argument except boolean is converted to real.
Generic Name	ABS
Remarks	Integer results are the same length, in bytes, as the argument.

ICHAR

Purpose	Returns the value of a character argument after conversion to an 8 byte integer.
Format	**ICHAR(a)**
Function	Specific
Argument type	Character
Result type	8-byte integer
Generic name	None
Remarks	The value returned depends on the collating weight of the argument in the collating sequence in use. For the ASCII collating sequence, the first character in the collating sequence is at position 0, the second character at position 1, the third at position 2, and so forth. For a user-specified collating sequence, two or more characters can have the same value. The argument is a character value with a length of one character, and the value returned is the integer position of that character in the collating sequence.

IDIM

Purpose	Returns the positive difference.
Format	**IDIM(a1,a2)**
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Same as the argument except boolean is converted to integer.
Generic name	DIM
Remarks	The result is a1-a2 if a1 is greater than a2, and zero if a1 is not greater than a2. Integer results are the same length, in bytes, as the argument.

IDINT

Purpose	Returns the value after conversion to integer.
Format	**IDINT(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	Integer.
Generic name	INT

IDNINT

Purpose	Returns the nearest integer.
Format	**IDNINT(a)**
Function	Specific
Argument type	Boolean or double precision
Result type	8-byte integer
Generic name	NINT
Remarks	The result is INT(a+.5) if a is positive or zero, and INT(a-.5) if a is negative.

IFIX

Purpose	Returns the value of the boolean or real argument after conversion to integer.
Format	**IFIX(a)**
Function	Specific
Argument type	Boolean or real
Result type	8-byte integer
Generic name	INT
Remarks	The result is INT(a).

INDEX

Purpose	Returns the location of substring a2 within string a1.
Format	**INDEX(a1, a2)**
Function	Specific
Argument type	a1,a2: character
Result type	8-byte integer
Remarks	If string a2 occurs as a substring within string a1, the result is an 8-byte integer indicating the starting position of the substring a2 within a1. If a2 does not occur as a substring within a1, the result is 0. If a2 occurs as a substring more than once within a1, only the starting position of the first occurrence is returned.

Control Data Extension

INSB

Purpose	Returns a copy of a4 with the bits from a1 inserted.
Format	**INSB(a1, a2, a3, a4)**
Function	Generic
Argument type	a1, a4: any data type except character a2, a3: byte or integer
Result type	Boolean
Remarks	The rightmost a3 bits of a1 are inserted at bit position a2 of a4. Argument a4 itself is not altered. Bits are numbered from the left starting with zero. Arguments a1 and a4 can be of any data type except character. Arguments a1 and a4 are converted to boolean using the BOOL function; the bits are extracted from BOOL(a1) and inserted into a copy of BOOL(a4). For integer or logical arguments, the length, in bytes, of the result is the same as the length of the argument. The result is undefined if a2 is less than zero or greater than 63 or if a3 is less than zero, or if (a2+a3) is greater than 64.

End of Control Data Extension

INT

Purpose	Performs type conversion to integer.
Format	**INT(a)**
Function	Generic
Argument type	Boolean, byte, integer, real, double precision, or complex
Result type	Integer
Specific names	INT For byte, real, and integer arguments only IFIX For real arguments only IDINT For double precision arguments only
Remarks	For a real or double precision argument, where |a1| < 1, the result is 0. Where |a1| ≥ 1, the result is the largest integer with the same sign as argument a that does not exceed |a|. For a complex argument, the real part is used and the result is the same as for a real argument. The result has the same length, in bytes as the argument. The result is an 8-byte integer for real, double precision, or complex arguments.

ISIGN

Purpose	Performs a transfer of sign.
Format	**ISIGN(a1, a2)**
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Same as the argument except boolean is converted to integer.
Generic name	SIGN
Remarks	For integer arguments, the result is the same length in bytes, as the argument. The result is |a1| if a2 is positive or zero, and -|a1| if a2 is negative.

Control Data Extension

LBOUND

Purpose	Returns the lower bound of dimension a2 of a1.
Format	**LBOUND(a1**,*a2*)
Function	Generic
Argument type	a1: any type a2: integer or byte
Result type	8-byte integer
Remarks	If you specify a2, the result is a scalar value. Argument a1 must be an array, array section, or array expression. Argument a2 must be a scalar value. If you omit a2, the lower bound of each dimension of a1 is returned in a one-dimensional array. The value of the ith element of the result of LBOUND(a) is the lower bound of the ith dimension of a1. The lower bound of an array section or array expression is defined to be 1. The result is undefined if a1 is an allocatable array that is not currently allocated.

End of Control Data Extension

LEN

Purpose	Returns the length of a character string.
Format	**LEN(a)**
Function	Specific
Argument type	Character
Result type	8-byte integer
Generic name	None

LGE

Purpose	Returns a logical value indicating lexically greater than or equal to.
Format	**LGE(a1, a2)**
Function	Specific
Argument type	Character
Result type	8-byte logical
Generic name	None
Remarks	The result is true if a1 follows a2 or a1 is equal to a2 in the ASCII collating sequence; the result is false otherwise. If the arguments are of unequal length, the shorter argument is treated as if it were extended on the right with blanks to the length of the longer argument.

LGT

Purpose	Return a logical value indicating lexically greater than.
Format	**LGT(a1, a2)**
Function	Specific
Argument type	Character
Result type	8-byte logical
Remarks	The result is true is a1 follows a2 in the ASCII collating sequence (shown in appendix C); the result is false otherwise. If the arguments are of unequal length, the shorter argument is treated as if it were extended on the right with blanks to the length of the longer argument.

LLE

Purpose	Returns a logical value indicating lexically less than or equal to.
Format	**LLE(a1, a2)**
Function	Specific
Argument type	Character
Result type	8-byte logical
Remarks	The result is true if a1 precedes a2 or a1 is equal to a2 in the ASCII collating sequence (shown in appendix C); the result is false otherwise. If the arguments are of unequal length, the shorter argument is treated as if it were extended on the right with blanks to the length of the longer argument.

LLT

Purpose	Returns a logical value indicating lexically less than.
Format	**LLT(a1, a2)**
Function	Specific
Argument type	Character
Result type	8-byte logical
Remarks	The result is true if a1 precedes a2 in the ASCII collating sequence (shown in appendix C); the result is false otherwise. If the arguments are of unequal length, the shorter argument is treated as if it were extended on the right with blanks to the length of the longer argument.

LOG

Purpose	Returns the natural logarithm (logarithm base e).
Format	**LOG(a)**
Function	Generic
Argument type	Boolean, complex, real, or double precision
Result type	Same as the argument except boolean is converted to integer.
Specific names	ALOG For real arguments only DLOG For double precision arguments only CLOG For complex arguments only

LOG10

Purpose	Returns a common logarithm (logarithm base 10).
Format	**LOG10(a)**
Function	Generic
Argument type	Boolean, real or double precision
Result type	Same as the argument except boolean is converted to integer.
Specific names	ALOG10 For real arguments only DLOG10 For double precision arguments only
Remarks	The argument must be greater than zero.

Control Data Extension

MASK

Purpose	Returns a boolean result.
Format	**MASK(a)**
Function	Specific
Argument type	Integer, byte, or boolean
Result type	Boolean
Remarks	The result is a word of **a** left-justified one bits followed by (64-**a**) zero bits.
	The result is undefined if a is less than zero or greater than 64.
	The argument is in the range 0 through 64.

End of Control Data Extension

Control Data Extension

MATMUL

Purpose	Returns the product of arguments a1 and a2.
Format	**MATMUL (a1, a2)**
Function	Generic
Argument type	a1,a2: any type but character.
Result type	Same as the argument except boolean is converted to integer.
Remarks	The two arguments must be multiplication compatible (the last dimension of a1 must have the same size as the first dimension of a2; both arguments must be of arithmetic type or both arguments must be of logical type). Neither argument can have more than two dimensions, and at least one must be two dimensional.

Both arguments must be arrays, array sections, or array expressions.

Integer or logical arguments must be the same length. The shape of the result array is determined as follows:

a1	a2	Result
(n,m)	(m,k)	(n,k)
(m)	(m,k)	(k)
(m,k)	(k)	(m)

End of Control Data Extension

MAX

Purpose: Returns the largest value from 2 through 500 arguments.

Format: **MAX(a,...,*a*)**

Function: Generic

Argument type: Boolean, byte, integer, real, or double precision

Result type: Same as the argument except boolean is converted to integer.

Specific names:

MAX0	For integer or byte arguments only
AMAX1	For real arguments only
DMAX1	For double precision arguments only

Remarks: All of the arguments that are not boolean must be of the same type.

Integer arguments must have the same length.

Control Data Extension

MAXVAL

Purpose: Returns the maximum element of a1 along dimension a2 corresponding to true elements of a3.

Format: **MAXVAL (a1,*a2*,*a3*)**

Function: Generic

Argument type:
a1: boolean, integer, byte, real or double
a2: boolean, integer, byte
a3: logical

Result type: Same as argument a1 except boolean is converted to integer.

Remarks: Argument a2 can be specified by the keyword DIM=.

Argument a3 is an array of with the same shape as a1 and can be specified with the keyword MASK=.

If argument a3 specifies no .TRUE. elements, the result is the largest machine number for the type of a1.

End of Control Data Extension

MAX0

Purpose	Returns the largest value from 2 through 500 arguments.
Format	**MAX0(a,**...*a*)
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Same as the argument except boolean is converted to integer.
Generic name	MAX
Remark	Integer arguments must be the same length.

MAX1

Purpose	Returns the largest value from 2 through 500 arguments.
Format	**MAX1(a,**...,*a*)
Function	Specific
Argument type	Boolean or real
Result type	8-byte integer
Generic name	None

Control Data Extension

MERGE

Purpose	Returns a result containing the values of a1 corresponding to true elements of a3, and the values of a2 corresponding to false elements of a3.
Format	**MERGE(a1, a2, a3)**
Function	Generic
Argument type	a1,a2: any type a3: 8-byte logical
Result type	Same as argument a1
Remarks	Argument a3 can be a scalar or array. If a3 is a scalar value, a2 and a1 must also be scalar values. If a3 is an array expression, a2 and a1 must also be array expressions. Arguments a1 and a2 must be of the same type and be conformable with a3. Logical or integer arguments a1 and a2 must be of the same length.

End of Control Data Extension

MIN

Purpose	Returns the smallest value from 2 through 500 arguments.
Format	**MIN(a,...,*a*)**
Function	Generic
Argument type	Boolean, byte, integer, real, or double precision
Result type	Same as the argument except boolean is converted to integer.
Specific names	MIN0 For byte or integer arguments only AMIN1 For real arguments only DMIN1 For double precision arguments only
Remarks	All the arguments must be of the same type and length unless they are boolean.

Control Data Extension

MINVAL

Purpose	Returns the minimum element of a1 along dimension a2 corresponding to true elements of a3.
Format	**MINVAL(a1**,*a2*,*a3*)
Function	Generic
Argument type	a1: boolean, integer, byte, real, or double precision a2: boolean, integer, or byte a3: logical
Result type	Same as the argument except boolean is converted to integer.
Remarks	Argument a2 can be specified by the keyword DIM=. Argument a3 is an array, array section, or array expression with the same shape as a1 and can be specified with the keyword MASK=. If argument a3 specifies no .TRUE. elements, the result is the smallest machine number for the type of a1.

End of Control Data Extension

MIN0

Purpose	Returns the smallest value from 2 through 500 arguments.
Format	**MIN0(a**,...*a*)
Function	Specific
Argument type	Boolean, byte, or integer
Result type	Same as the argument except boolean is converted to integer.
Generic name	MIN
Remarks	All integer arguments must be the same length; the result is the same length as the arguments.

MIN1

Purpose	Returns the smallest value from 2 through 500 arguments.
Format	**MIN1(a,...,*a*)**
Function	Specific
Argument type	Boolean or real
Result type	8-byte integer
Generic name	None

MOD

Purpose	Returns a1 modulus a2 (the remainder of a1 divided by a2).
Format	**MOD(a1, a2)**
Function	Generic
Argument type	Boolean, byte, integer, real, or double precision
Result type	Same as the argument except boolean is converted to integer.
Specific names	AMOD For real arguments only DMOD For double precision arguments only MOD For byte or integer arguments only
Remarks	The result for integer arguments is the same length as the argument. If only one argument is boolean, the result is the type of the other argument. If both arguments are boolean, the result is integer. The result is a1-(INT(a1/a2)*a2). If a2 is zero, results are undefined.

Control Data Extension

NEQV

Purpose	Returns the nonequivalence of 2 through 500 arguments.
Format	**NEQV(a,...,*a*)**
Function	Specific
Argument type	Any type but character
Result type	Boolean
Generic name	None
Remarks	The result is the same as for the boolean exclusive or (.NEQV.) operator.

End of Control Data Extension

NINT

Purpose	Returns the nearest integer.
Format	**NINT(a)**
Function	Generic
Argument type	Boolean, real or double precision
Result type	8-byte integer
Specific names	NINT For real arguments only IDNINT For double precision arguments only
Remarks	If the argument is zero or positive, the result in (INT(a+.5)). If the argument is negative, the result is (INT(a-.5)).

Control Data Extension

OR

Purpose	Returns the boolean sum of 2 through 500 arguments.
Format	**OR(a,...,***a***)**
Function	Specific
Argument type	Any type but character
Result type	Boolean
Generic name	None
Remarks	The result is the same as for the boolean .OR. operator.

PACK

Purpose	Returns a one-dimensional array consisting of all elements of a1 corresponding to true elements of a2.
Format	**PACK(a1,a2,***a3***)**
Function	Generic
Argument type	a1,a3: any type a2: logical
Result type	Same type as a1 except boolean is converted to integer.
Remarks	The arguments are taken in subscript order. Argument a1 and a2 must be conformable arrays, array sections or array expressions. Integer and logical arguments can be of any length, but arguments a2 and a3 must be of the same length. The optional argument a3 is a size specification for the result array; it must be a one-dimensional array of the same type as a1 with at least as many elements as there are true elements in a2. If a3 is omitted, the size of the result is the number of true elements in a2. Arguments a1 and a3 must be the same type and length.

End of Control Data Extension

Control Data Extension

PRODUCT

Purpose	Returns the product of elements in argument a1 along dimension a2 corresponding to .TRUE. elements of a3.
Format	**Product(a1,a2,***a3***)**
Function	Generic
Argument type	a1: boolean, integer, byte, real, double precision, or complex a2: boolean, integer, or byte a3: logical
Result type	Same as the argument except boolean is converted to integer.
Remarks	If a2 is omitted, the function is applied to all elements of a1 corresponding to .TRUE. elements of a3 to yield a scalar value; else the result has the same shape as a1 without dimension a2.

Argument a1 must be an array, array section, or array expression.

Argument a2 can be specified by the keyword DIM=. The optional argument a3 is an array, array section, or an array expression with the same shape as argument a1. Each .TRUE. value in argument a3 specifies the corresponding element that is included in the product.

Argument a3 can be specified by the keyword MASK=. If a3 is omitted, an array conforming to a1 and having all elements .TRUE. is used. If argument a3 specified no .TRUE. elements, the result is one.

End of Control Data Extension

Control Data Extension

PTR

Purpose	Returns the address of a parameter to a C language subprogram.
Format	**PTR(a)**
Function	Generic
Argument type	Any
Result type	Boolean
Remarks	This function can only be used in a statement that is calling a routine written in the C language. The result cannot be used within a FORTRAN program unit. The argument must be a variable or an array.

RANF

Purpose	Returns a random number.
Format	**RANF(***a1***)**
Function	Specific
Argument type	Any type
Result type	Real
Generic name	None
Remarks	Successive calls to RANF yield a random sequence of numbers. Each value returned is real and is in the range 0<result<1. You can reinitialize the seed by calling the RANSET function described in Chapter 10. If an array-valued result is desired, the shape of the argument specifies the shape of the result. If you do not specify an argument, a scalar value is returned.

End of Control Data Extension

Control Data Extension

RANK

Purpose	Returns the number of dimensions in a1.
Format	**RANK(a1)**
Function	Generic
Argument type	Any type
Result type	8-byte integer
Remark	Argument a1 can be a scalar, array, array section, or array expression. If a1 is a scalar, the result is zero. If a1 is an array section, the result is the number of section selectors in the array section reference. If a1 is an allocatable array, the function returns the number of dimensions of the argument whether or not a1 is allocated.

End of Control Data Extension

REAL

Purpose	Performs type conversion and returns a real result.
Format	**REAL(a)**
Function	Generic
Argument type	Boolean, byte, integer, real, double precision, or complex
Result type	Real
Specific names	FLOAT For integer or byte arguments only SNGL For double precision arguments only REAL For integer, real, or complex arguments only
Remarks	A boolean argument is treated as a bit string and is not changed. For an 8-byte integer or double precision argument, REAL(a) has as much precision of the significant part of the argument as a real item can contain. For a complex argument (ar,ai), the result is ar. Byte and 2- or 4-byte integer arguments are converted to boolean values using BOOL(a) before being passed to REAL.

Control Data Extension

SEQ

Purpose: Returns a one-dimensional array.

Format: **SEQ(a1, a2,** *a3*)

Function: Generic

Argument type: a1, a2, a3: 8-byte integer

Result type: 8-byte integer

Remarks: The result consists of :

a1+a3, a1+2*a3 ... a1+(s−1)*a3

where s is MAX((a2−a1+a3)/a3, 1).

Arguments a1 and a3 must be scalar values. Argument a3 must be a nonzero scalar value.

If a3 is omitted, the value 1 is used.

End of Control Data Extension

Control Data Extension

SHAPE

Purpose	Returns the size of dimension a2 in a1.
Format	**SHAPE(a1,** *a2*)
Function	Generic
Argument type	a1: any type a2: boolean, integer or byte
Result type	8-byte integer
Remarks	Argument a1 must be an array, array section, or array expression.

Logical or integer arguments can be of any length; however, arguments a1 and a2 must be of the same length. Argument a2 is an optional dimension specification and must be a scalar value.

If you specify a2, the result is a scalar value.

If you omit a2, the size of each dimension of a1 is returned in a one-dimensional array. The value of the ith element of the result is the size of the ith dimension of a1.

The result of SHAPE(a1) is undefined if a1 is an assumed-size array.

End of Control Data Extension

Control Data Extension

SHIFT

Purpose	Returns a shifted result.
Format	**SHIFT(a1, a2)**
Function	Specific
Argument type	a1: any type but character a2: byte, integer or boolean
Result type	Boolean
Generic name	None
Remarks	The boolean result is a1 shifted a2 bit positions. The shift is left circular if a2 is positive, or right with sign extension and end off if a2 is negative. Argument a2 is in the range −64 through +64. If a2 is outside this range, the result is undefined.

End of Control Data Extension

SIGN

Purpose	Returns a value after transfer of sign.
Format	**SIGN(a1, a2)**
Function	Generic
Argument type	Integer, byte, boolean, real, or double precision
Result type	Same as the argument except boolean is converted to integer.
Specific names	SIGN For real arguments only DSIGN For double precision arguments only ISIGN For byte or integer arguments only
Remarks	The result is $\lvert a1 \rvert$ if a2 is zero or positive. The result is $-\lvert a1 \rvert$ if a2 is negative. The result is the same length, in bytes, as the argument.

SIN

Purpose	Returns the sine of an argument.
Format	**SIN(a)**
Function	Generic
Argument type	Real, double precision, complex, or boolean
Result type	Same as the argument except boolean is converted to real.
Specific names	SIN For real arguments only DSIN For double precision arguments only CSIN For complex arguments only
Remarks	The argument is in radians.

Control Data Extension

SIND

Purpose	Returns the sine of a boolean or real argument.
Format	**SIND(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	None
Remarks	The argument is in degrees.

End of Control Data Extension

SINH

Purpose	Returns the hyperbolic sine of an argument.
Format	**SINH(a)**
Function	Generic
Argument type	Real, double precision, or boolean
Result type	Same as the argument except boolean is converted to real.
Specific names	SINH For real arguments only DSINH For double precision arguments only

Control Data Extension

SIZE

Purpose	Returns the size.
Format	**SIZE(a)**
Function	Generic
Argument type	Any type
Result type	8-byte integer
Generic name	None
Remarks	Argument a1 must be an array, array section, or array expression. Integer and logical arguments can be of any length. The result is a scalar value. The result of SIZE is undefined if argument a is an assumed-size array or an allocatable array that is not currently allocated.

End of Control Data Extension

SNGL

Purpose	Returns the value of an argument after conversion to single precision real.
Format	**SNGL(a)**
Function	Specific
Argument type	Boolean or double precision.
Result type	Real.
Generic name	REAL

SQRT

Purpose	Returns a principal square root.
Format	**SQRT(a)**
Function	Generic
Argument type	Real, double precision, complex, or boolean
Result type	Same as the argument except boolean is converted to real.
Specific names	SQRT For real arguments only DSQRT For double precision arguments only CSQRT For complex arguments only
Remarks	The argument must not be negative.

Control Data Extension

SUM

Purpose: Returns the sum of elements in argument a1 along dimension a2 corresponding to .TRUE. elements in a3.

Format: **SUM(a1,** *a2, a3)*

Function: Generic

Argument type:
a1: byte, real, double precision, integer, complex, or boolean
a2: boolean, byte, integer
a3: logical

Result type: Same as argument a1 except boolean is converted to integer.

Remarks: Argument a1 must be an array, array section, or array expression. The result is the same length as the argument. The optional argument a2 specifies which dimension is to be used for the summation to occur along. If a2 is omitted, the function is applied to all elements of a1 corresponding to true elements of a3 to yield a scalar value; else the result has the same shape as a1 without dimension a2.

The dimension (argument 2) can be specified by the keyword DIM=. The optional argument a3 is an array, array section, or array expression with the same shape as a1. Each true value specifies the corresponding element in a1 that is to be summed. Argument a3 can be specified by the keyword MASK=. If a3 is omitted, an array conforming to a1 and having all elements .TRUE. is used. If argument a3 specifies no .TRUE. elements, the result is zero.

End of Control Data Extension

Control Data Extension

SUM1S

Purpose	Returns the number of bits that are set.
Format	**SUM1S(a)**
Function	Generic
Argument type	Boolean, complex, double precision, 8-byte integer or real
Result type	8-byte integer
Remarks	A set bit is one with the binary value 1. If the argument is of type double precision or complex, only the first word is used.

End of Control Data Extension

TAN

Purpose	Returns the tangent.
Format	**TAN(a)**
Function	Generic
Argument type	Boolean, real, or double precision
Result type	Same as the argument except boolean is converted to real.
Specific names	TAN For real arguments only DTAN For double precision arguments only
Remarks	The argument is in radians.

Control Data Extension

TAND

Purpose	Returns the tangent of an argument.
Format	**TAND(a)**
Function	Specific
Argument type	Boolean or real
Result type	Real
Generic name	None
Remarks	The argument is in degrees.

End of Control Data Extension

TANH

Purpose	Returns the hyperbolic tangent.
Format	**TANH(a)**
Function	Generic
Argument type	Boolean, real, or double precision
Result type	Same as the argument except boolean is converted to real.
Specific names	TANH For real arguments only DTANH For double precision arguments only

Control Data Extension

UBOUND

Purpose	Returns the upper bound of dimension a2 of a1.
Format	**UBOUND(a1,** *a2*)
Function	Generic
Argument type	a1: any type a2: boolean, integer or byte
Result type	8-byte integer
Remarks	Argument a1 must be an array, array section, or array expression. Argument a2 is an optional dimension specification and must be a scalar value. If you specify a2, the result is a scalar value. If you omit a2, the upper bound of each dimension of a1 is returned in a one-dimensional array. The value of the ith element of the result of UBOUND(a1) is the upper bound of the ith dimension of a1. The result is undefined if a1 is an allocatable array that is not currently allocated. The result of UBOUND(a1) is undefined if a1 is an assumed-size array. The result of UBOUND(a1, a2) is undefined if a1 is an assumed-size array and a2 specifies the last dimension. The upper bound of an array section is the number of elements in the dimension.

End of Control Data Extension

Control Data Extension

UNPACK

Purpose	Returns an array with the same shape as a3 and the same type as a1.
Format	**UNPACK(a1, a2, a3)**
Function	Generic
Argument type	a1: any type a2: logical a3: any type
Result type	Same as the argument except boolean is converted to integer.
Remarks	Arguments a1 and a3 must be the same type.

Elements of the result array corresponding to true elements of a2 are taken in subscript order and assigned successive values of a1. Each element of the result array corresponding to false elements of a2 is assigned the value of the corresponding element of a3.

Argument a1 must be a one-dimensional array of the same type as a3. Argument a2 must be an array of type logical (any length) and conformable to a3. Argument a3 must be an array of the same type as argument a1 and the same shape as a2. The number of elements in a1 must be greater than or equal to the number of true values in a2. Logical and integer arguments a1 and a3 must be of the same length. If a3 is a scalar value, each element of the result array corresponding to a false value of a2 is assigned the value of a3.

End of Control Data Extension

Control Data Extension

XOR

Purpose	Returns the exclusive OR of 2 through 500 arguments.
Format	**XOR(a,** ... ,*a*)
Function	Specific
Argument type	Any type but character
Result type	Same as the argument except boolean is converted to integer.
Generic name	None
Remarks	The result is the same as for the boolean exclusive or (.XOR.) operator.

End of Control Data Extension

System Command Language and Utility Subprograms 10

NOS/VE Status Subprograms 10-2
CONDNAM 10-4
CONDSYM 10-5
FLP$FORMAT_MESSAGE 10-6
INTCOND 10-9
UPKSTAT 10-10

Communicating with NOS/VE 10-11
Execution Command Parameters 10-12
Defining Execution Command Parameters 10-13
Execution Command Parameters Subprograms 10-14
GETBVAL 10-15
GETCVAL 10-16
GETIVAL 10-17
GETSCNT 10-18
GETSVAL 10-19
GETVCNT 10-20
GETVREF 10-21
SCLKIND 10-22
TSTPARM 10-23
TSTRANG 10-24
SCL Variable Subprograms 10-25
ABORT 10-26
CREV 10-27
DELV 10-28
REDBVAR 10-29
REDCVAR 10-30
REDIVAR 10-31
REDSVAR 10-32
WRTBVAR 10-33
WRTCVAR 10-34
WRTIVAR 10-35
WRTSVAR 10-36
CALL SCLCMD Statement 10-37

Utility Subprograms 10-38
Random Number Generation 10-40
RANSET 10-40
RANGET 10-40
Debugging Subprograms 10-41
DUMP and PDUMP 10-41
STRACE 10-41
Error Handling Subprograms 10-42
LEGVAR 10-42
SYSTEM 10-43
SYSTEMC 10-44
LIMERR and NUMERR 10-47
Collating Sequence Control Subprograms 10-48
COLSEQ 10-49
WTSET 10-50
CSOWN 10-51

Date and Time Information Subprograms 10-52
DATE 10-52
JDATE 10-52
TIME or CLOCK 10-52
SECOND 10-52
Miscellaneous Utility Subprograms 10-53
DISPLA 10-53
REMARK 10-53
SSWTCH 10-53
EXIT 10-54
CHGUCF 10-54
DEFAUA 10-57
Input/Output-Related Subprograms 10-58

NOS/VE and Utility Subprograms 10

FORTRAN provides a set of subprograms that enable you to take advantage of various NOS/VE features. The subprograms and their purposes are:

- NOS/VE Status Subprograms

  Process a variable containing a NOS/VE status variable.

- NOS/VE Communication Subprograms

  Control program execution command parameters and allow you to access System Command Language values used before or after program execution. You can also execute any NOS/VE SCL command within a FORTRAN program using the CALL SCLCMD statement.

- Utility Subprograms

  Perform random number generation, error handling, collating sequence control, date and time retrieval, and others.

Within each of the three categories of subprograms, the subprograms are described in alphabetical order.

NOS/VE Status Subprograms

The NOS/VE status subprograms allow you to manipulate the NOS/VE status variable from within your FORTRAN program. The NOS/VE status variable contains three fields of information:

NORMAL field (1 byte)

Contains a boolean flag indicating if an error occurred.

CONDITION field (5 bytes)

Contains the condition code, which is the binary combined value of the product identifier and the condition number.

TEXT field (256 bytes)

Contains information for the text of the message.

The NOS/VE status variable has a record structure that is incompatible with FORTRAN data types and forms. Because of this, you should not try to print or access it without using the status subprograms described in this section. The NOS/VE status variable information is returned to a FORTRAN program in the form of a character variable or array.

If a FORTRAN character variable is used, it must be declared as CHARACTER*264. If an array is used, it must be declared with 264 elements of CHARACTER*1. These provide for a 262-character variable plus a 2-character pad to maintain word alignment.

The subprograms process the FORTRAN variable or array and allow you to manipulate the status variable fields in the FORTRAN string or array. The subprograms and their brief descriptions are:

CONDNAM

Returns the condition name from an integer condition code or FORTRAN condition number.

CONDSYM

Returns the string representation for a condition code in the form of product identifier and condition number.

FLP$FORMAT_MESSAGE

Returns a NOS/VE formatted message.

INTCOND

Returns the integer value of the condition code.

UPKSTAT

Returns the value of the NORMAL field, the product identifier and condition number from the CONDITION field, the length of the TEXT field, and the value in the TEXT field.

NOTE

The routines GETSVAL and REDSVAR, also described in this chapter, return the status variable information from an existing SCL status variable outside your program. The NOS/VE Status Processing subprograms described in this section process status variables that are returned directly to a FORTRAN program. For example, the status variable is returned directly from the CREV, DELV, or SCLCMD subroutines, a system-resident CYBIL routine, or from the *iostat* specifier on input/output statements.

The NOS/VE Status subprograms are described in alphabetical order.

CONDNAM

The CONDNAM function returns the condition name from an integer representation of the CONDITION field of a NOS/VE status variable. The function has the form:

CONDNAM (condition)

condition

Integer expression specifying the value of the condition code.

The value returned by the CONDNAM function is a character string with a length of 31. CONDNAM must be declared type CHARACTER*31 in the calling program. If the specified condition number does not exist, the string 'UNKNOWN_CONDITION' is returned. If the specified condition number does exist, the condition name is returned.

You can use the INTCOND function to first return an integer representation of the condition code. For example:

```
CHARACTER CONDITION*31, CONDNAM*31, S1*264
  :
CALL CREV ('BALANCE', 'INTEGER', LEN, 1, 1, 'JOB', S1)
IF (INTCOND(S1) .NE. 0) THEN
   IF (CONDNAM(INTCOND(S1)) .EQ. 'CLE$VAR_ALREADY_CREATED')THEN
     :
   ENDIF
  :
ENDIF
```

The call to the CREV subroutine creates an SCL integer variable named BALANCE, with a scope of JOB. After execution, the variable S1 contains the NOS/VE status information. If the value returned by the INTCOND function is not zero, then an error occurred on the CALL CREV statement. The second IF statement uses CONDNAM to check if the error that occurred is CLE$VAR_ALREADY_CREATED.

Condition names and their associated condition codes are listed in the NOS/VE Diagnostic Messages manual. To read a description of a condition name online, specify the condition name on the NOS/VE HELP command. For example, to read about the condition CLE$UNKNOWN_VARIABLE in the online MESSAGES manual, enter

```
/help manual=messages subject='cle$unknown_variable'
```

The online MESSAGES manual provides a brief description of the condition, the associated product identifier and condition number, and a recommended user action.

CONDSYM

The CONDSYM call returns the string representation of an integer condition code. The string representation is in the form of product identifier and condition number. This call has the form:

CALL CONDSYM (condition, string, len)

condition

Integer expression specifying either the condition code or FORTRAN condition number whose value is to be returned as a string.

string

Character variable to receive the string representation. A size of 6 is usually long enough for FORTRAN conditions. A value shorter than the argument is left justified and blank filled; a longer value is truncated on the right.

len

Integer variable to receive the length of the returned string.

The value returned in **string** is normally of the form *xx number*, where *xx* is the product identifier portion of the condition code and *number* is the condition number portion of the condition code. If the identifier portion of the condition code is not 2 displayable characters, the string represents the condition code as a number of up to 12 digits.

CONDSYM can be used with values returned in the *iostat* input/output specifier.

If a FORTRAN condition number is supplied for **condition**, the returned product identifier is the string FL.

The INTCOND function can be used first to return an integer representation of the condition code.

This example shows how to use INTCOND to return an integer representation of the condition code and CONDSYM to return a string representation:

```
CHARACTER STRING*6
INTEGER INTCOND, CONDITION, LEN, STAT
  :
CALL CREV('BALANCE', 'INTEGER', LEN, 1, 1, 'JOB', STAT)
IF (INTCOND(STAT) .NE. 0) THEN
    CALL CONDSYM(INTCOND(STAT), STRING, LEN)
    IF (STRING .EQ. 'CL 490') THEN
       PRINT *, 'Variable BALANCE already exists.'
    ENDIF
ENDIF
```

FLP$FORMAT_MESSAGE

The FLP$FORMAT_MESSAGE call returns a formatted NOS/VE message using the message template information of a NOS/VE status variable. The FLP$FORMAT_MESSAGE call has the form:

CALL FLP$FORMAT_MESSAGE (msgstat, msglev, msglcnt, msgltxt, msgllen, fstat)

msgstat

Character variable specifying the NOS/VE status information to be formatted into a message. It must be declared with a length of 264 or greater.

The **msgstat** parameter should be a value returned in the *fstat* parameter of the CREV, DELV, SCLCMD, or FLP$FORMAT_MESSAGE statements, a CYBIL OST$STATUS variable, or the *iostat* specifier on input/output statements.

msglev

Character expression specifying the level of detail in the message. Options are:

'CURRENT' ('C' or 'OSC$CURRENT_MESSAGE_LEVEL')

The message level is the current setting in the job.

'BRIEF' ('B' or 'OSC$BRIEF_MESSAGE_LEVEL')

The message appears without the product identifier and condition code.

'FULL' ('F' or 'OSC$FULL_MESSAGE_LEVEL')

The message appears with the product identifier and condition code.

Options specified in lowercase are also acceptable.

The product identifier and condition code are needed to locate the message in the NOS/VE Diagnostic Messages manual.

The current message level used in the job can be changed by the CHANGE_MESSAGE_LEVEL command as described in the NOS/VE System Usage manual.

msglcnt

Integer variable specifying the maximum number of message lines in the **msgltxt** array. This variable receives the count of the actual number of lines in the **msgltxt** array on return from FLP$FORMAT_MESSAGE.

msgltxt

Character array to receive the formatted message. Each array element value determines the maximum length of the corresponding message line. This value must be from 32 to 256 characters. The unused portion of a line is blank-filled. The actual length of a line without trailing blanks is returned in the corresponding element of the **msgllen** array.

The number of elements in the **msgltxt** array should be large enough to accommodate the number of lines typically expected. The number of array elements that actually receive message lines is returned in the **msglct** msglcnt variable. Elements beyond the number of total lines in the formatted message are not filled.

CONDSYM

The CONDSYM call returns the string representation of an integer condition code. The string representation is in the form of product identifier and condition number. This call has the form:

CALL CONDSYM (condition, string, len)

condition

Integer expression specifying either the condition code or FORTRAN condition number whose value is to be returned as a string.

string

Character variable to receive the string representation. A size of 6 is usually long enough for FORTRAN conditions. A value shorter than the argument is left justified and blank filled; a longer value is truncated on the right.

len

Integer variable to receive the length of the returned string.

The value returned in **string** is normally of the form *xx number*, where *xx* is the product identifier portion of the condition code and *number* is the condition number portion of the condition code. If the identifier portion of the condition code is not 2 displayable characters, the string represents the condition code as a number of up to 12 digits.

CONDSYM can be used with values returned in the *iostat* input/output specifier.

If a FORTRAN condition number is supplied for **condition**, the returned product identifier is the string FL.

The INTCOND function can be used first to return an integer representation of the condition code.

This example shows how to use INTCOND to return an integer representation of the condition code and CONDSYM to return a string representation:

```
CHARACTER STRING*6
INTEGER INTCOND, CONDITION, LEN, STAT
  :
CALL CREV('BALANCE', 'INTEGER', LEN, 1, 1, 'JOB', STAT)
IF (INTCOND(STAT) .NE. 0) THEN
    CALL CONDSYM(INTCOND(STAT), STRING, LEN)
    IF (STRING .EQ. 'CL 490') THEN
       PRINT *, 'Variable BALANCE already exists.'
    ENDIF
ENDIF
```

FLP$FORMAT_MESSAGE

The FLP$FORMAT_MESSAGE call returns a formatted NOS/VE message using the message template information of a NOS/VE status variable. The FLP$FORMAT_MESSAGE call has the form:

CALL FLP$FORMAT_MESSAGE (msgstat, msglev, msglcnt, msgltxt, msgllen, fstat)

msgstat

Character variable specifying the NOS/VE status information to be formatted into a message. It must be declared with a length of 264 or greater.

The **msgstat** parameter should be a value returned in the *fstat* parameter of the CREV, DELV, SCLCMD, or FLP$FORMAT_MESSAGE statements, a CYBIL OST$STATUS variable, or the *iostat* specifier on input/output statements.

msglev

Character expression specifying the level of detail in the message. Options are:

'CURRENT' ('C' or 'OSC$CURRENT_MESSAGE_LEVEL')

The message level is the current setting in the job.

'BRIEF' ('B' or 'OSC$BRIEF_MESSAGE_LEVEL')

The message appears without the product identifier and condition code.

'FULL' ('F' or 'OSC$FULL_MESSAGE_LEVEL')

The message appears with the product identifier and condition code.

Options specified in lowercase are also acceptable.

The product identifier and condition code are needed to locate the message in the NOS/VE Diagnostic Messages manual.

The current message level used in the job can be changed by the CHANGE_MESSAGE_LEVEL command as described in the NOS/VE System Usage manual.

msglcnt

Integer variable specifying the maximum number of message lines in the **msgltxt** array. This variable receives the count of the actual number of lines in the **msgltxt** array on return from FLP$FORMAT_MESSAGE.

msgltxt

Character array to receive the formatted message. Each array element value determines the maximum length of the corresponding message line. This value must be from 32 to 256 characters. The unused portion of a line is blank-filled. The actual length of a line without trailing blanks is returned in the corresponding element of the **msgllen** array.

The number of elements in the **msgltxt** array should be large enough to accommodate the number of lines typically expected. The number of array elements that actually receive message lines is returned in the **msglct** msglcnt variable. Elements beyond the number of total lines in the formatted message are not filled.

msgllen

Integer array to receive the lengths of the message lines returned in the **msgltxt** array. Each element in the **msgllen** array specifies the length of the line in the corresponding element of the **msgltxt** array. The **msgllen** array must have at least the same number of elements as the **msgltxt** array.

fstat

Character variable of size 264 to receive the status resulting from the execution of the FLP$FORMAT_MESSAGE call; **fstat** must be declared with a length of 264 or more in the calling program.

FLP$FORMAT_MESSAGE uses the condition code in **msgstat** and the message level specified to find the relevant message template. If a message template exists for the condition code, FLP$FORMAT_MESSAGE returns the existing message as a series of lines with each line corresponding to one element of **msgltxt**. The actual length of each message line (without trailing blanks) is contained in the corresponding element of **msgllen**. The total number of lines in the formatted message is returned in **msglcnt**.

If a message line is longer than the maximum length indicated by the corresponding element in the **msgltxt** array, FLP$FORMAT_MESSAGE formats the message into more than one line. It splits the message at a space, if possible; otherwise, it appends two periods to the end of the line to indicate continuation to the next line. Any unprintable character is represented by a question mark (?).

If **msgltxt** is not long enough to contain all lines in the message, *fstat* contains the NOS/VE status for FLE$TRUNCATED_MESSAGE. Only the lines specified in **msglcnt** on the call to FLP$FORMAT_MESSAGE are returned in **msgltxt**. The value returned in **msglcnt** is the actual number of lines that would have been in the complete message. This lets you know how much of the message is missing and allows you to select a more appropriate array size.

Example:

```
      PROGRAM FORMAT_MESSAGE
      PARAMETER (MAXLLEN=75)
      PARAMETER (N=6)
      CHARACTER STATUS_VAR*264, FSTAT*264, MSGLTXT(N)*(MAXLLEN),
     + CONDNAM*31
      INTEGER MSGLCNT, MSGLLEN(N)
      CALL SCLCMD ( 'ATTF $USER.MY_FILE', STATUS_VAR)
      MSGLCNT=N
      CALL FLP$FORMAT_MESSAGE (STATUS_VAR,'FULL', MSGLCNT, MSGLTXT,
     + MSGLLEN, FSTAT)
      IF (CONDNAM(INTCOND(FSTAT)).EQ.'FLE$TRUNCATED_MESSAGE') MSGLCNT=N
      DO 10 I=1, MSGLCNT
        PRINT *, MSGLTXT(I)(1:MSGLLEN(I))
   10 CONTINUE
      END
```

The example uses the status information in STATUS_VAR to format a message. The message text will have a maximum of 6 lines, with each line having a maximum length of 75 characters.

The generated message has a maximum of 6 lines, with each line having a maximum length of 75 characters. The PRINT statement prints the message, removing trailing blanks from each line.

If no message template exists for the specified condition code, the content of the status variable is returned within the following line:

```
CC=xx number TEXT=string
```

CC is the condition code for the status record formed by joining xx (the product identifier) and number (the condition number).

NOS/VE messages are listed in the NOS/VE Diagnostic Messages manual. User message templates can be defined and placed in the command list to be searched. Information on defining message templates is in the NOS/VE Object Code Management manual under Message Module Generator Utility and in the CYBIL for NOS/VE System Interface manual.

INTCOND

The INTCOND function returns the condition code as an integer value. This function has the form:

INTCOND (fstat)

fstat

Character variable, array, or substring specifying the name of the FORTRAN variable containing the NOS/VE status information for which the condition code is to be returned. It must be of size 264 or larger.

The **fstat** parameter is defined with the value returned in the **fstat** parameter of the CREV, DELV, SCLCMD, or FLP$FORMAT_MESSAGE statements, the value of a CYBIL OST$STATUS variable, or the *iostat* specifier on input/output statements.

The value returned is of type integer. If the NORMAL field of the status variable is TRUE, (indicating no error or exception), the value returned is zero. Otherwise, the value returned is the condition code specifying the exception.

INTCOND must be typed (by default or otherwise) as integer.

You can use the CONDNAM function to return the associated condition name of a value returned by the INTCOND function. For example:

```
CHARACTER CONDITION*31, CONDNAM*31, S1*264
    :
CALL CREV('BALANCE', 'INTEGER', LEN, 1, 1, 'JOB', S1)
CONDITION=CONDNAM(INTCOND(S1))
```

The call to the CREV Subroutine specifies the creation of an SCL integer variable name BALANCE, with a scope of JOB. After execution, the variable CONDITION contains the condition name. If the variable BALANCE does not already exist, it is created and the value of CONDITION is FLE$NO_ERROR. If the variable BALANCE does already exist, the value of CONDITION is CLE$VAR_ALREADY_CREATED.

Condition names and their associated condition codes are listed in the Diagnostic Messages for NOS/VE manual.

UPKSTAT

The UPKSTAT call returns the information contained in a NOS/VE status variable in separate parts. This call has the form:

CALL UPKSTAT (fstat, norm, id, cond, len, text)

fstat

Character variable, array, or substring specifying the name of the FORTRAN variable containing the NOS/VE status information. It must be of size 264 or larger.

The **fstat** parameter is defined with the value returned in the **fstat** parameter of the CREV, DELV, SCLCMD, or FLP$FORMAT_MESSAGE statements, the value of a CYBIL OST$STATUS variable, or the *iostat* specifier on input/output statements.

norm

Logical variable to receive the value of the NORMAL field. The NORMAL field contains one of the values TRUE or FALSE.

id

Character variable to receive the value of the product identifier of the CONDITION field. Should be declared with at least a length of 2.

cond

Integer variable to receive the condition code.

len

Integer variable to receive the length (number of characters) of the data in the TEXT field.

text

Character variable to receive the value of the TEXT field. Should be 256 characters in length. A value shorter than the argument is left justified and blank filled; a longer value is truncated on the right.

If the value returned in the **norm** parameter is true, the **id**, **cond**, **len**, and **text** parameters are undefined.

Communicating with NOS/VE

The System Command Language (SCL) is the language by which you communicate to NOS/VE. There are three ways a FORTRAN program can communicate with NOS/VE and SCL:

- You can pass values to and from NOS/VE by specifying parameters on the execution command and by using the Parameter Interface Subprograms.
- The SCLCMD call can execute any NOS/VE command from within your program.
- The variable interface subprograms can retrieve values of SCL variables and store values into SCL variables.

Execution Command Parameters

An execution command can specify a list of parameters to be used for communicating with NOS/VE. The parameters can be specified positionally or by parameter name and must be separated by a space or a comma.

Parameters specified by name have the following form:

name=value

where name is the parameter name and value is the parameter value. The parameter value can be specified as:

- A single value element.
- A value set, consisting of a series of value elements separated by commas or spaces and enclosed in parentheses.
- A value list, consisting of a series of value sets separated by commas or spaces and enclosed in parentheses.

A value element is either a single value or a value range represented by a lower bound and an upper bound separated by two periods:

lower-bound .. upper-bound

The most common way of specifying a parameter value is by a single value element.

Each SCL parameter value is defined to have a specific kind. The valid value kinds are FILE, NAME, STRING, INTEGER, BOOLEAN, STATUS, and ANY. The kind of a parameter value must match the kind of value defined for that parameter in the C$ PARAM directive.

With the exception of the predefined $PRINT_LIMIT and STATUS parameters and parameters to be used for file name substitution, each SCL parameter that appears on the execution command must be defined in the source program by the C$ PARAM directive described in the next section. The $PRINT_LIMIT and STATUS parameters and the method of file name substitution are described in chapter 12, Compilation and Execution.

Examples:

```
/lgo par=3
```

The value 3 is specified for the parameter PAR.

```
/lgo p1=((1, 2, 3)  (4, 5, 6))
```

A value list is specified for parameter P1. The list contains two value sets, and each value set contains three value elements.

```
/var
var/v: string
var/varend
/v='afile'
  :
/lgo fp=v
```

The string variable V is specified for the parameter FP.

Using the parameter interface subprograms, you can get information about SCL parameters specified on the execution command including whether or not a particular parameter is present, the type of the parameter, and the parameter values.

Defining Execution Command Parameters

The C$ PARAM directive defines parameters that are to be specified on the execution command. The general form of the C$ PARAM directive is

C$ PARAM (pdefs)

pdefs

A character constant expression, whose value is of the form **'pdef**; *...; pdef'*, where **pdef** is a valid SCL parameter definition. The format for parameter definitions is described in the NOS/VE System Usage manual.

The characters C$ must appear in positions 1 and 2, and the string PARAM must begin in or after position 7. The C$ PARAM directive cannot be continued on a subsequent line. Long parameter definitions can be made by specifying the required character constant in one or more PARAMETER statements, each of which can be continued over 19 lines, and then referencing the symbolic name of the constant in the C$ PARAM directive. (See the second example below).

Any parameter that appears on the execution command must have been defined by a C$ PARAM directive in the program to be executed. The C$ PARAM directive defines such parameter properties as:

- Parameter name
- Whether or not the parameter is required on the execution command
- Parameter default values
- Parameter kind
- Allowable number of value sets
- Allowable number of value elements in a value set

A parameter specified on the execution command must conform to its definition in the C$ PARAM directive.

Only one C$ PARAM directive can be specified in a program, and it must appear in the main program. It can be placed anywhere after the PROGRAM statement and can define one or more parameters. A C$ PARAM directive in a subprogram has no effect.

If a program contains a C$ PARAM directive, the PROGRAM statement should contain only the program name; no file equivalencing can be done.

Example:

```
C$    PARAM('A:INTEGER; B:VAR OF STRING')
```

This directive defines two parameters. Values specified for parameter A will be of kind integer; and values specified for parameter B will be SCL variables of kind string.

The following example shows how to represent five parameters with a single C$ PARAM directive:

```
              :
         CHARACTER*(*) P1, P2, P3, P4, P5
         PARAMETER(P1='P1:BOOLEAN;')
         PARAMETER(P2='P2:LIST 1 .. 7, 2 .. 2 RANGE OF INTEGER  3 .. 10;')
         PARAMETER(P3='P3:RANGE OF VAR OF STATUS;')
         PARAMETER(P4='P4:LIST 2 .. 4 OF FILE;')
         PARAMETER(P5='P5:RANGE OF INTEGER')
   C$    PARAM (P1//P2//P3//P4//P5)
              :
```

Execution Command Parameters Subprograms

The execution command parameter subprograms retrieve parameter names, parameter values, and other information.

A separate subprogram is provided for retrieving each of the different parameter kinds. You must use the call that corresponds to the kind of parameter you wish to access. Each subprogram call returns a single value element of a parameter value.

You communicate with SCL through arguments specified in the subprogram calls.

Arguments of type integer must be 8-byte integers; that is, typed as INTEGER*8 or INTEGER. Each call requires you to specify certain information about the parameter. Information you need to specify depends on how the parameter was defined in the C$ PARAM directive. This information includes:

- Parameter name
- Value set number of the desired value (1 if parameter is defined with a single element)
- Value number of the desired value (1 if parameter is defined with a single element)
- For range parameters, whether the value is the upper or lower range bound.

For subprogram arguments in which character values are returned, values shorter than the argument length are left-justified and blank-filled; longer values are truncated on the right.

The parameter interface subprogram calls are described on the following pages in alphabetical order. All parameters to the subprogram calls are required except those in italics. Parameters either contain a value that you are specifying or receive a value on return from the subprogram.

GETBVAL

The GETBVAL call returns the value of an SCL boolean parameter. The SCL boolean type corresponds to the FORTRAN type logical. This call has the form:

CALL GETBVAL (param, setnum, valnum, lhi, val, log)

param

Character expression specifying the name of the SCL boolean parameter whose value is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum

Integer expression specifying the value set number of the requested value. Specify 1 if value sets are not defined for the parameter.

valnum

Integer expression specifying the value number within the value set of the requested value. Specify 1 if multiple values are not defined for the parameter.

lhi

For range parameters, a character expression specifying whether the requested value is the lower or upper range bound. Options are: 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH' regardless of whether or not the parameter is defined as a range.

val

Variable to receive the value of the specified boolean parameter. It must be declared type logical.

log

Integer variable to receive a number indicating how the boolean value was specified. One of the following values is returned:

0 Value specified as TRUE or FALSE.
1 Value specified as YES or NO.
2 Value specified as ON or OFF.

You can use this value to determine the appropriate form to use for responses to the boolean parameter.

GETCVAL

The GETCVAL call returns the value of a STRING, NAME, KEY, or FILE parameter. This call has the form:

CALL GETCVAL (param, setnum, valnum, lhi, len, val)

param

Character expression specifying the name of the STRING, FILE, KEY, or NAME parameter whose value is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum

Integer expression specifying the value set number of the requested value. Specify 1 if value sets are not defined for the parameter.

valnum

Integer expression specifying the value number within the value set of the requested value. Specify 1 if multiple values are not defined for the parameter.

lhi

For range parameters, character expression specifying whether the requested value is the lower or upper range bound. Options are: 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH', regardless of whether or not a parameter is defined as a range.

len

Integer variable to receive the length of the requested value.

val

Character variable to receive the value of the specified parameter. Up to 256 characters are returned. Values shorter than the argument are left justified and blank-filled; longer values are truncated on the right. The variable must be at least 256 characters long to receive FILE parameter values.

GETIVAL

The GETIVAL call returns the value of an SCL integer parameter. This call has the form:

CALL GETIVAL (param, setnum, valnum, lhi, val, rad)

param

Character expression specifying the name of the SCL integer parameter whose value is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum

Integer expression specifying the value set number of the value to be returned. Specify 1 if value sets are not defined for the parameter.

valnum

Integer expression specifying the value number within the value set of the value to be returned. Specify 1 if multiple values are not defined for the parameter.

lhi

For range parameters, a character expression specifying whether the requested value is the lower or upper range bound. Options are: 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH', regardless of whether or not the parameter is defined as a range.

val

Integer variable to receive the value of the SCL integer parameter.

rad

Integer variable to receive the base (radix) of the integer value. One of the values 2, 8, 10, or 16 is returned. The radix returned is the one specified in the last assignment to the variable; if none was specified, 10 is returned. You can use this value to determine the appropriate radix to use for responses to the integer parameter.

GETSCNT

The GETSCNT function returns an integer indicating the number of value sets specified for the parameter. The function reference has the form:

GETSCNT (param)

param

Character expression specifying the name of the parameter for which the number of value sets is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

GETSCNT must be declared type integer in the calling program.

GETSVAL

The GETSVAL call returns the values of an existing SCL status variable. This call has the form:

CALL GETSVAL (param, setnum, valnum, lhi, norm, id, cond, len, text)

param

Character expression specifying the name of the SCL status parameter for which status values are to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum

Integer expression specifying the value set number of the value to be returned. Specify 1 if value sets are not defined for the parameter.

valnum

Integer expression specifying the value number within the value set of the value to be returned. Specify 1 if multiple values are not defined for the parameter.

lhi

Character expression specifying whether the requested value is the lower or upper range bound. Options are: 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH' regardless of whether or not a range is defined for the parameter.

norm

Logical variable to receive the value (TRUE or FALSE) of the NORMAL field. If nml is TRUE, the remaining fields are undefined.

id

Character variable to receive the value of the ID portion of the status variable. Should be at least two characters long.

cond

Integer variable to receive the value of the CONDITION field of the status variable.

len

Integer variable to receive the length (characters) of the value returned in the text argument.

text

Character variable to receive the value of the TEXT field of the status variable. Up to 256 characters are returned. A value shorter than the argument is left justified and blank-filled; a longer value is truncated on the right.

GETVCNT

The GETVCNT function returns the number of values in the specified value set of the specified parameter. The function reference has the form:

GETVCNT (param, setnum)

param

Character expression specifying the name of the parameter for which the number of values in the specified value set is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum

Integer expression specifying the value set number for which the number of values is to be returned.

GETVCNT must be declared type integer in the calling program.

GETVREF

The GETVREF call returns a variable reference specified for a parameter. This call has the form:

CALL GETVREF (param, setnum, valnum, lhi, ref, knd, lbnd, ubnd, len)

param

Character expression specifying the name of the parameter for which the variable reference is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum

Integer expression specifying the value set number of the requested variable reference. Specify 1 if value sets are not defined for the parameter.

valnum

Integer expression specifying the value number within the value set of the requested variable. Specify 1 if multiple values are not defined for the parameter.

lhi

Character expression specifying whether the requested value is the lower or upper range value. Options are: 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH' regardless of whether or not a range is defined for the parameter.

ref

Character variable to receive the variable reference as it appears on the execution command. A value shorter than the argument is left justified and blank-filled; a longer value is truncated on the right.

knd

Character variable to receive a string indicating the kind (type) of variable. Contains one of the strings 'STRING', 'INTEGER', 'BOOLEAN', or 'STATUS'.

lbnd

Integer variable to receive the lower bound of a variable for which a range is defined.

ubnd

Integer variable to receive the upper bound of a variable for which a range is defined.

len

Integer variable to receive the length of a string variable.

The argument **ref** contains contains the variable name in character format. This variable name can be passed to the variable interface routines, which can retrieve or alter the value of the variable.

SCLKIND

The SCLKIND call returns a string indicating the SCL kind (type) of a parameter. This call has the form:

CALL SCLKIND (param, setnum, valnum, kind)

param

Character expression specifying the name of the parameter for which the kind (type) is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum

Integer expression specifying the value set number of the parameter value.

valnum

Integer expression specifying the value number within the value set of the parameter value.

kind

Character variable to receive a string indicating the parameter kind. One of the following values is returned:

'FILE'	'STATUS'
'NAME'	'STRING'
'INTEGER'	'BOOLEAN'
'ANY'	

TSTPARM

The TSTPARM function returns the logical value .TRUE. if the specified parameter appeared on the execution command. Otherwise, TSTPARM returns the value .FALSE. The TSTPARM function reference has the form:

TSTPARM (param)

param

Character expression specifying the name of the parameter to be tested. If the length exceeds 31 characters, the excess characters are truncated on the right.

TSTPARM must be declared type logical in the calling program.

TSTRANG

The TSTRANG function determines whether a parameter value was specified as a range. The function returns the logical value .TRUE. if the named value was specified on the execution command as a range. Otherwise, TSTRANG returns the logical value .FALSE. The TSTRANG function reference has the form:

TSTRANG (param, setnum, valnum)

param

Character expression specifying the name of the parameter to be tested. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum

Integer expression specifying the value set number of the value to be tested. Specify 1 if value sets are not defined for the parameter.

valnum

Integer expression specifying the value number within the value set of the value to be tested. Specify 1 if multiple values are not defined for the parameter.

TSTRANG must be declared type logical.

SCL Variable Subprograms

The SCL variable subprogram calls are used to retrieve or alter the values of existing SCL variables and to define new SCL variables. New variables must be defined by a call to the CREV subroutine before they can be referenced by the variable interface calls. (SCL variables referenced by the variable interface calls need not be specified on the execution command.)

For SCL variables specified as parameter values on the execution command, the GETVREF parameter interface call retrieves variable names, which you specify in subsequent variable interface calls to identify the variable you wish to access.

NOTE

The variable interface subprogram calls do not use the C$ PARAM directive.

There are two types of variable interface calls: calls that retrieve the value of a variable (routine names that begin with letters RED) and calls that store a value into a variable (routine names that begin with letters WRT). In addition, the CREV routine is used to define a new variable, and the DELV routine deletes the definition of a variable.

A separate subroutine is provided for each variable kind. When using the variable interface calls to access an SCL variable, you must use the call that corresponds to the kind of that variable.

SCL variables can be defined as scalar variables or arrays. In the following subprograms, the arguments val, bool, len, rad, dig, exponen, norm, id, cond, and text can be variables or arrays, but must be declared with enough locations to receive a value corresponding to each element of the requested SCL variable. When a variable interface subprogram specifies an SCL array, the entire array is referenced; you cannot reference individual elements within an SCL array. Variables or arrays of type integer must be 8-byte integers, that is, typed as INTEGER*8 or INTEGER.

The variable interface subprograms are described below in alphabetical order. All parameters to the subprogram calls are required except those in italics.

ABORT

The ABORT call sets the fields of the variable specified by the STATUS parameter on the execution command and terminates program execution. This call has the form:

CALL ABORT (id, cond, text)

id

Character expression whose value is to be placed in the ID field of the STATUS parameter. If more than two characters are specified, excess characters are truncated on the right.

cond

Integer expression whose value is to be placed in the CONDITION field of the STATUS parameter. It must be in the range 0 through 999999.

text

Character expression whose value is to be stored in the TEXT field of the STATUS parameter. If more than 256 characters are specified, excess characters are truncated on the right.

CREV

The CREV call defines a new SCL variable. This call has the form:

CREV (var, knd, len, lb, ub, scope, *fstat*)

var

Character expression specifying the name of the SCL variable being defined. SCL variable names can be any combination of alphanumeric characters, underscores, and other special characters as long as it does not begin with a number and contains 31 or fewer characters. For a table of allowed special characters, see SCL Names in the NOS/VE System Usage manual.

knd

Character expression specifying the SCL kind (type) of the variable. One of the strings 'STRING', 'INTEGER', 'BOOLEAN', or 'STATUS'.

len

For a type STRING variable, an integer expression specifying the length of the variable elements. Its maximum value is 256. Although it is not used for a non-STRING variable, a value must be specified.

lb

Integer expression defining the lower bound of the array being defined. Specify 1 if a scalar variable is being defined.

ub

Integer expression defining the upper bound of the array being defined. Specify 1 if a scalar variable is being defined.

scope

Character expression defining the scope of the variable. One of the following values:

'LOCAL'

Variable is created local to the block.

'XDCL'

Variable is created with the externally declared (XDCL) attribute.

'XREF'

Variable is created with the externally referenced (XREF) attribute.

'JOB'

Variable is created in the job block with the XDCL attribute.

'name'

Variable is created in the utility block specified by name, with the XDCL attribute.

fstat

Character variable to receive the status resulting from the execution of the CALL CREV command; *fstat* must be declared with a length of 264 in the calling program. Use the NOS/VE Status subprograms to retrieve data from this variable.

DELV

The DELV call removes the definition of an SCL variable defined by a previous CREV call. It cannot delete a variable defined outside the program unit. This call has the form:

CALL DELV (var, *fstat*)

var

Character expression specifying an SCL variable name.

fstat

Character variable to receive the status resulting from the execution of the CALL DELV command; *fstat* must be declared with a length of 264 in the calling program. Use the NOS/VE Status subprograms to retrieve data from the variable.

If a program unit contains a DELV call, the CREV call that creates the variable must be in the same program unit.

REDBVAR

The REDBVAR call returns the values of an SCL boolean variable (type logical in FORTRAN). This call has the form:

CALL REDBVAR (ref, dim, val, bool)

ref

Character expression specifying an SCL boolean variable or array element reference.

dim

Integer expression specifying the dimension of **val**. Specify 1 if **val** is a variable.

val

Logical variable or array to receive the values of the boolean variable. If **ref** specifies an SCL array, **val** should contain enough elements to receive a value for each array element.

bool

Integer variable or array indicating how the boolean variable was specified; each word receives one of the following numbers:

0	Variable value was specified as TRUE or FALSE.
1	Variable value was specified as YES or NO.
2	Variable value was specified as ON or OFF.

One value is returned for each value of the requested variable.

REDCVAR

The REDCVAR call returns the values of an SCL string variable. This call has the form:

CALL REDCVAR (ref, dim, len, val)

ref

Character expression specifying the name of the SCL string variable whose value is to be returned. It must be an SCL variable or array element reference.

dim

Integer expression specifying the dimension of **val**. Specify 1 if **val** is a variable.

len

Integer variable or array to receive the length of the SCL string variable. If **ref** specifies an SCL array, **len** must contain enough elements to receive a value for each array element.

val

Character variable or array to receive the value of the SCL variable. If **ref** specifies an SCL array, **val** should contain enough elements to receive a value for each element of **ref**. Each value can be up to 256 characters long. Values shorter than the length of **val** are left-justified and blank-filled. Longer values are truncated on the right.

REDIVAR

The REDIVAR call returns the values of an SCL integer variable. This call has the form:

CALL REDIVAR (ref, dim, vals, rad)

ref

Character expression specifying the name of the SCL integer variable for which values are to be returned. It must be an SCL variable or array element reference.

dim

Integer expression specifying the dimension of **val**. Specify 1 if **val** is a variable.

val

Integer variable or array to receive the value of the SCL variable; if **ref** specifies an SCL array, **val** must contain enough elements to receive all of the array values.

rad

Integer variable or array to receive the base (radix) of the SCL integer variable. If **ref** specifies an SCL array, **rad** must contain enough elements to receive a value for each value of the array. Each base is one of 2, 8, 10, or 16. The radix returned is the one specified in the last assignment to the SCL variable; if none was specified, 10 is returned.

REDSVAR

The REDSVAR call returns the values of an existing SCL Status variable. This call has the form:

CALL REDSVAR (ref, dim, norm, id, cond, text)

ref

Character expression specifying the name of the SCL Status variable for which a value is to be returned. It must be an SCL variable or array element reference.

dim

Integer expression specifying the dimension of the **norm**, **id**, **cond**, and **text** arrays; specify 1 if **norm**, **id**, **cond**, and **text** are variables.

norm

Logical variable or array to receive the value of the NORMAL field of the status variable. If **ref** specifies an SCL array, **norm** must contain enough elements to receive a value for each element of **ref**. If **norm** is TRUE for a particular value, the corresponding ID, CONDITION, and TEXT fields are undefined.

id

Character variable or array to receive the value of the ID field of the status variable. If **ref** specifies an SCL array, **id** must contain enough elements to receive a value for each element of **ref**. Each element of **id** should be at least two characters long.

cond

Integer variable or array to receive the value of the CONDITION field of the status variable. If **ref** specifies an SCL array, **cond** must contain enough elements to receive a value for each element of **ref**.

text

Character variable or array to receive the value of the TEXT field of the status variable. If **ref** specifies an SCL array, **text** must contain enough elements to receive a value for each element of **ref**. Up to 256 characters can be returned for each element of **ref**. Values shorter than the length of the **text** argument are left-justified and blank-filled. Longer values are truncated on the right.

WRTBVAR

The WRTBVAR call stores values into the specified SCL boolean variable or array. This call has the form:

CALL WRTBVAR (ref, dim, val, bool)

ref

Character expression specifying the SCL boolean variable or array for which values are to be stored. It must be an SCL variable or array element reference.

dim

Integer expression specifying the dimension of **val**; specify 1 if **val** specifies a single value.

val

Logical expression specifying the value or values to be stored. If **ref** is an array, **val** must be an array containing one value for each element or **ref**.

bool

Integer expression indicating how the boolean values are specified. The indicators are:

0 Values specified as TRUE or FALSE.
1 Values specified as YES or NO.
2 Values specified as ON or OFF.

If **val** is an array, **bool** must be an array containing one value for each element of **val**.

SCL type boolean corresponds to FORTRAN type logical.

WRTCVAR

The WRTCVAR call stores values into the specified SCL string variable or array. This call has the form:

CALL WRTCVAR (ref, dim, len, str)

ref

Character expression specifying the name of the SCL string variable or array into which values are to be stored. It must be an SCL variable or array element reference.

dim

Integer expression specifying the dimension of **str**; specify 1 if **str** specifies a single value.

len

Integer expression specifying the lengths of the elements of **str**. If **str** is an array, **len** must be an array containing one value for each element of **str**.

str

Character expression whose value is a string to be stored. If **ref** specifies an SCL array, **str** must be an array containing one element for each value to be stored in **ref**. SCL strings can be up to 256 characters long.

WRTIVAR

The WRTIVAR call stores values into the specified SCL integer variable or array. This call has the form:

CALL WRTIVAR (ref, dim, val, rad)

ref

Character expression specifying the SCL integer variable or array into which values are to be stored. It must be an SCL variable or array element reference.

dim

Integer expression specifying the dimension of **val**; specify 1 if **val** specifies a single value.

val

Integer expression specifying the values to be stored in the SCL variable or array. If **ref** specifies an array, **val** must be an array containing one value for each element of **ref**.

rad

Integer expression specifying the base (radix) of the values to be stored. If **val** is an array, **rad** must be an array containing a radix for each value being stored.

WRTSVAR

The WRTSVAR call stores values into the specified SCL status variable or array. This call has the form:

CALL WRTSVAR (ref, dim, norm, id, cond, text)

ref

Character expression specifying the name of the status variable or array into which values are to be stored. It must be an SCL variable or array element reference.

dim

Integer expression specifying the dimension of **norm**, **id**, **cond**, and **text**; specify 1 if **norm**, **id**, **cond**, and **text**; specify single values.

norm

Logical expression specifying the value to be stored in the NORMAL field. If **ref** is an SCL array, **norm** must be an array containing one value for each element of **ref**. Each value must be specified as TRUE or FALSE.

If TRUE is specified for a particular value, values specified for the corresponding **id**, **cond**, and **text** arguments need not be defined.

id

Character expression specifying the value to be stored in the ID field. If **ref** is an SCL array, **id** must be an array containing one value for each element of **ref**.

cond

Integer expression specifying the value to be stored in the CONDITION field. If **ref** specifies an array, **cond** must be an array containing one value for each element of **ref**.

text

Character expression specifying the value to be stored in the TEXT field. If **ref** specifies an array, **text** must be an array containing one value for each element of **ref**. Up to 256 characters can be stored in a TEXT field.

CALL SCLCMD Statement

The SCLCMD call specifies a command string that is passed to NOS/VE and executed. This call has the form:

CALL SCLCMD (text, *fstat***)**

text

Character expression specifying a valid NOS/VE command. The maximum length of **text** is 256 characters. The expression **text** can contain multiple commands separated by semicolons.

fstat

Character variable to receive the status resulting from the execution of the NOS/VE command; *fstat* must be declared with a length of 264 in the calling program. Use the NOS/VE Status subprograms to retrieve data from the variable.

Example:

```
CALL SCLCMD('SETFA FILE=ABC MAXIMUM_RECORD_LENGTH=500')
```

This call executes the SET_FILE_ATTRIBUTE command and sets the maximum record length attribute of file ABC.

Example:

```
 CHARACTER COMMAND*50, STATUS_OF_COMMAND*264
 READ *, COMMAND
 CALL SCLCMD(COMMAND, STATUS_OF_COMMAND)
 IF(INTCOND(STATUS_OF_COMMAND) .NE.
+ 0) PRINT * 'ERROR ON COMMAND'
```

This example uses a character variable to read a NOS/VE command from the terminal. The status variable information is returned in STATUS_OF_COMMAND. The INTCOND function is used to check for an error.

Example:

```
 CALL SCLCMD('DISPLAY_VALUE $FILE
+ (AFILE,FILE_ORGANIZATION)', FSTAT)
```

This example shows how to execute an SCL function from a FORTRAN program. The SCLCMD call executes a DISPLAY_VALUE command which references the $FILE function. The $FILE function displays the value of the FILE_ORGANIZATION attribute of a file named AFILE. The NOS/VE status information is returned in the FSTAT, which is a character variable of length 264.

Utility Subprograms

The utility subprograms described in the following paragraphs are supplied by the FORTRAN library. A user-supplied subprogram with the same name as a library subprogram overrides the library subprogram.

Arguments that are of type integer must be 8-byte integers; that is, typed as INTEGER or INTEGER*8. Table 10-1 presents a summary of the FORTRAN utility subprograms:

Table 10-1. FORTRAN Utility Subprograms

Category of Subprogram	Subprogram Name	Description
Random Number Generation	RANSET	Initializes RANF function.
	RANGET	Returns current seed of RANF function.
Debugging	DUMP	Produces a memory dump and terminates program.
	PDUMP	Produces a memory dump and returns control to program.
	STRACE	Produces a subprogram traceback.
Error Handling	LEGVAR	Tests a variable for an infinite or indefinite value.
	SYSTEM	Issues a runtime error message.
	SYSTEMC	Alters internal error processing specifications.
	LIMERR	Inhibits program termination for errors caused by invalid input data.
	NUMERR	Returns number of errors that have occurred since last LIMERR call.
Collating Sequence Control	COLSEQ	Selects a collating weight table.
	WTSET	Modifies a collating weight table.
	CSOWN	Defines a collating sequence.
Input/Output Status	UNIT	Checks status of BUFFER IN or BUFFER OUT.
	EOF	Tests for end-of-file.
	LENGTH and LENGTHX	Returns number of words in the last record read.
	LENGTHB	Returns number of bytes in the last record read.
	IOCHEC	Tests for parity error.

(Continued)

Table 10-1. FORTRAN Utility Subprograms *(Continued)*

Category of Subprogram	Subprogram Name	Description
Mass Storage Input/Output	OPENMS	Opens a random access file.
	WRITMS	Writes a record to a random access file.
	READMS	Reads a record from a random access file.
	CLOSMS	Closes a random access file.
	STINDX	Specifies a subindex for a random access file.
Miscellaneous Input/Output	MOVLEV	Moves a block of noncharacter data from one area of memory to another.
	MOVLCH	Moves a block of character data from one area of memory to another.
	CONNEC	Connects a file to the terminal.
	DISCON	Disconnects a file from the terminal.
Date and Time Subprograms	DATE	Returns the current date.
	JDATE	Returns the current Julian date.
	TIME	Returns the current clock time.
	SECOND	Returns elapsed time (CPU seconds) since beginning of job.
	CLOCK	Returns the current clock time.
Miscellaneous Utility Subprograms	DISPLA	Places a name and value in the job log.
	REMARK	Places a message in the job log.
	SSWTCH	Tests the system sense switches.
	EXIT	Terminates program execution.
	CHGUCF	Enables and disables system conditions.
	DEFAUA	Enables application billing.

Random Number Generation

The following subprograms are used in conjunction with the RANF intrinsic function to control the generation of random numbers.

RANSET

The RANSET call specifies the starting value (seed) for the RANF function. This call has the form:

CALL RANSET (n)

n

Initial (seed) value; an 8-byte integer, real, or boolean expression

A subsequent call to RANF uses the value to calculate a random number.

RANSET may alter the seed value you supply; thus, the value returned by a subsequent call to RANGET may differ from the value you specified in a RANSET call.

RANGET

The RANGET obtains the current seed of RANF between 0 and 1. This call has the form:

CALL RANGET (n)

n

Real, 8-byte integer, or boolean variable or array element to receive the current seed value.

The value returned can be passed to RANSET at a later time to regenerate the same sequence of random numbers.

The value returned should not be used for any other purpose.

Debugging Subprograms

FORTRAN provides several subprograms to aid in the debugging process. You can also use the Debug utility to debug your programs. Appendix E provides an introduction to the Debug utility.

DUMP and PDUMP

The DUMP call and the PDUMP call produce a memory dump on file $OUTPUT in the requested format. These calls have the forms:

CALL DUMP (a, b, f, ..., *a*, *b*, *f*)

CALL PDUMP (a, b, f, ..., *a*, *b*, *f*)

a

Variable or array element of any type that is the first word of the area to be dumped.

b

Variable or array element of any type that is the last word of the area to be dumped.

f

Decimal integer specifying the format of the dump:

0 Hexadecimal dump
1 Real dump
2 Integer dump
3 Octal dump

DUMP and PDUMP are identical, except that PDUMP returns control to the calling program and DUMP terminates program execution. The maximum number of occurrences of the triplet **a**, **b**, **f** is 20. The number of occurrences of the triplet **a**, **b**, **f** can vary throughout a program, but a warning message is issued.

The first and last word specified in a PDUMP call must be in the same segment.

STRACE

The STRACE call provides traceback information. This call has the form:

CALL STRACE

The traceback begins with the subroutine calling STRACE and continues through the chain of calling subroutines until the main program is reached. Traceback information is written to the file OUTPUT.

Error Handling Subprograms

The following subprograms provide error handling capabilities.

LEGVAR

The LEGVAR function checks the value of a type real variable to determine whether the variable contains an indefinite or infinite (out-of-range) value. The LEGVAR function reference has the form:

LEGVAR (a)

a

Real variable whose value is to be checked.

LEGVAR returns one of the following values:

−1	Variable contains an indefinite value.
0	Variable contains a valid value.
1	Variable contains an infinite value.

LEGVAR is of type integer.

Debugging Subprograms

FORTRAN provides several subprograms to aid in the debugging process. You can also use the Debug utility to debug your programs. Appendix E provides an introduction to the Debug utility.

DUMP and PDUMP

The DUMP call and the PDUMP call produce a memory dump on file $OUTPUT in the requested format. These calls have the forms:

CALL DUMP (a, b, f, ..., *a, b, f*)

CALL PDUMP (a, b, f, ..., *a, b, f*)

a

Variable or array element of any type that is the first word of the area to be dumped.

b

Variable or array element of any type that is the last word of the area to be dumped.

f

Decimal integer specifying the format of the dump:

0 Hexadecimal dump
1 Real dump
2 Integer dump
3 Octal dump

DUMP and PDUMP are identical, except that PDUMP returns control to the calling program and DUMP terminates program execution. The maximum number of occurrences of the triplet **a, b, f** is 20. The number of occurrences of the triplet **a, b, f** can vary throughout a program, but a warning message is issued.

The first and last word specified in a PDUMP call must be in the same segment.

STRACE

The STRACE call provides traceback information. This call has the form:

CALL STRACE

The traceback begins with the subroutine calling STRACE and continues through the chain of calling subroutines until the main program is reached. Traceback information is written to the file OUTPUT.

Error Handling Subprograms

The following subprograms provide error handling capabilities.

LEGVAR

The LEGVAR function checks the value of a type real variable to determine whether the variable contains an indefinite or infinite (out-of-range) value. The LEGVAR function reference has the form:

LEGVAR (a)

a

Real variable whose value is to be checked.

LEGVAR returns one of the following values:

−1	Variable contains an indefinite value.
0	Variable contains a valid value.
1	Variable contains an infinite value.

LEGVAR is of type integer.

SYSTEM

The SYSTEM call enables you to issue a FORTRAN runtime error message at any time during program execution. This call has the form:

CALL SYSTEM (errnum, msg)

errnum

Error code. (An integer expression whose value is in the range 0 through 9999 decimal.)

msg

Error message in the form of a character constant.

The error message is issued immediately when SYSTEM is called. If the specified error number corresponds to one of the FORTRAN runtime error numbers, then the error is forced to occur and the normal FORTRAN error message header, including the error number, is written along with the specified message to file $ERRORS.

Error numbers used by FORTRAN retain the severity associated with them. Error numbers 51 (nonfatal) and 52 (fatal) are reserved for your use. If an error number is greater than the highest number defined for FORTRAN, 52 is substituted. If error number zero is specified, the message is ignored and control is returned to the calling program. Each line is printed unless the sum of lines written to $OUTPUT and $ERRORS exceeds the print limit, in which case the job is terminated.

FORTRAN runtime error messages, and associated error condition numbers, are listed in the NOS/VE Diagnostic Messages manual.

Example:

`CALL SYSTEM (3, 'CHECK DATA')`	The FORTRAN error 3 header, the message CHECK DATA, and a traceback are printed immediately upon execution of the CALL SYSTEM statement.

SYSTEMC

The SYSTEMC call enables you to alter error processing. This call has the form:

CALL SYSTEMC (errnum, slist, recov)

errnum

Integer expression whose value determines the FORTRAN error number for which nonstandard recovery is to be implemented. The value must be in the range 0 through 9999. Only FORTRAN runtime library error numbers are supported.

Some expressions in the source program generate a reference to an external library of mathematical routines on the system called the Math Library. For Math Library errors, the equivalent FORTRAN error number for the error must be used.

slist

Six-element integer array containing the following error processing specifications:

element 1	1=fatal; 0=nonfatal.
element 2	Print frequency.
element 3	Frequency increment.
element 4	Print limit.
element 5	Recovery routine selector: 0=Recovery routine not provided. 1=Recovery routine provided.
element 6	Maximum traceback limit applicable to all errors. Default is 20.

recov

Optional name of user-written recovery routine. (Routine name must be declared in EXTERNAL statement.) If the routine is used to process errors from a Math Library routine, the routine should not change common block items.

These specifications are ignored for erroneous data input from a connected (terminal) file.

The following error processing operations can be controlled by SYSTEMC:

- Print frequency. The default print frequency value is zero. If the value is changed to n by a call to SYSTEMC, diagnostic and traceback information is listed every nth time until the print limit is reached.

- Frequency increment. The default frequency increment is 1. This specification can be changed by a call to SYSTEMC if the call specifies the print frequency as zero. If the frequency increment is zero, diagnostic and traceback information is not listed; if it is 1, such information is listed until the print limit is reached; if it is set to n, information is listed only the first n times unless the print limit is reached first.

- Print limit. The default print limit is 10.

- Severity level. The severity levels for an error are fatal and nonfatal. All errors that you define using the listed numbers in a SYSTEM call retain the indicated severity. However, the severity of any FORTRAN error can be changed by a call to SYSTEMC.

- Recovery selector. Specifies whether or not a recovery routine is provided. A user-specified error recovery routine activated by a call to SYSTEMC can be canceled by a subsequent call with element 5 of slist set to zero.

- Maximum traceback limit. The default value is 20. If this value is changed to n by a call to SYSTEMC, then a traceback prints the first n routines in the traceback chain are printed.

A negative value for any element in the slist array indicates that the current value of that specification is not to be changed.

If SYSTEMC has been called, an error summary is issued at job termination indicating the number of times each error occurred since the first call to SYSTEMC. If elements 2, 3, and 4 of slist are all specified as zero in the last SYSTEMC call executed for an error number, then the error summary for that error number is not be issued at job termination.

Except for an error detected by a Math Library function, a user-supplied error recovery routine should be a subroutine subprogram. If a user-supplied error recovery routine is used for input/output error processing, it should not perform input/output operations.

For an error detected by a Math Library function, a user-supplied error recovery routine should be a function subprogram of the same type as the function detecting the error; the arguments of the functions must agree in number and type.

SYSTEMC should not be used to recover from errors in a vector version of a Math Library function. Vector versions are used when programs are compiled with VECTORIZATION_LEVEL=HIGH on the VFORTRAN command or the intrinsic function reference contains an array object as an argument. (To use the scalar version of a Math Library function in a program that is compiled with VECTORIZATION_LEVEL=HIGH, declare the function external by using the EXTERNAL statement.)

For more information about diagnostic messages, see the NOS/VE Diagnostics Messages manual.

When an error previously referenced by a SYSTEMC call is detected, the following sequence of operations is initiated:

1. Diagnostic and traceback information is printed in accordance with the internal error processing specifications as modified by the SYSTEMC call. The traceback information is terminated by any of the following conditions:

   Calling routine is a main program.
   Maximum traceback limit is reached.

2. If a nonfatal error occurs for which a SYSTEMC call has provided a recovery routine, control returns to the routine that called the routine detecting the error.

3. If the error is fatal, the job is terminated.

4. An error summary is printed at job termination, except for errors whose last SYSTEMC call had elements 2, 3, and 4 of slist set to zero.

Example:

```
EXTERNAL ERRFN
DIMENSION IRAY(6)
DATA IRAY/6* 1/
IRAY(4)=0
IRAY(5)=1
CALL SYSTEMC(62, IRAY, ERRFN)
       :
```

These statements suppress printing of error message 62 and transfer control to a user-defined recovery routine named ERRFN.

LIMERR and NUMERR

The LIMERR call and NUMERR function reference enable you to input data to a program without the risk of termination when improper data is input. These calls have the forms:

CALL LIMERR (lim)

NUMERR ( )

lim

Eight-byte integer expression whose value is the limit for the number of errors.

When LIMERR is called, the program does not terminate when data errors are encountered until the number of errors occurring after the call exceeds the value of the argument **lim**.

LIMERR can be used to inhibit job termination when data is input with a formatted, NAMELIST, internal file, or list directed read or with DECODE statements. LIMERR has no effect on the processing of errors in data input from a terminal file since you can correct those interactively.

LIMERR initializes an error count and specifies a maximum limit on the number of data errors allowed before termination. The specified limit continues in effect for all subsequent READ statements until the limit is reached. LIMERR can be reactivated with another call, which reinitializes the error count and resets the limit. A LIMERR call with **lim** specified as zero nullifies a previous call; improper data then results in job termination as usual.

When improper data is encountered in a formatted or namelist READ (or in a DECODE statement) with LIMERR in effect, the bad data field is bypassed, and processing continues at the next field. When improper data is encountered in a list directed READ, control transfers to the statement immediately following the READ statement.

The NUMERR function returns the number of data errors that have occurred since the last LIMERR call. The previous error count is lost when LIMERR is called, and the count is reinitialized to zero. NUMERR is of type integer (8 bytes).

The following example illustrates the use of LIMERR and NUMERR:

```
CALL LIMERR (200)
READ (1,125) (A(I), I=1, 1500)
IF (NUMERR() .GT. 0) THEN
   CALL LIMERR (200)
      :
END IF
READ (1, 125) (B(I), I=1, 1500)
```

When LIMERR is called, a limit of 200 errors is established and the error count is set to zero. After A is read, NUMERR() is checked. If no errors occurred during the read, control transfers to the statement following ENDIF. If errors occurred, LIMERR reinitializes the error count and the rest of the block IF is executed.

Had LIMERR not been called, any invalid data encountered during the read operation would have caused a fatal error.

Collating Sequence Control Subprograms

Character relational expressions are evaluated according to a collating sequence determined by a collation weight table. A collation weight table is a one-dimensional integer array 256 words long with a lower bound of zero. Each element of the weight table has a value between zero and 255 inclusive. The 256 words correspond to the 256 characters of the ASCII character set. The collating weight for the character with the hexadecimal character code of i is the value of the element i of the weight table. The ASCII characters, the corresponding character codes, and the available weight tables are given in appendix C.

The value of a weight table element need not be unique within the table; that is, several characters can have the same collating weight.

You can use a default collating sequence, select one of the predefined collating sequences, or define your own collating sequence. The default collating sequence is the ASCII (fixed) collating sequence.

To select one of the predefined collating sequences or define your own collating sequence, you must specify the DEFAULT_COLLATION=USER parameter on the VECTOR_FORTRAN command, or precede the first character comparison in the program by a C$ COLLATE (USER) directive. Either of these specifications automatically selects the OSV$DISPLAY64_FOLDED collating sequence. You can subsequently select one of the other predefined tables by calling the COLSEQ routine, modify any of the predefined tables by calling the WTSET routine, or define your own table by calling the CSOWN routine.

The relational operations .EQ., .LT., .LE., .GT., .NE., and .GE., and the intrinsic functions CHAR and ICHAR use the collating sequence currently in effect as established by the following:

- The DEFAULT_COLLATION parameter on the VECTOR_FORTRAN command
- The C$ COLLATE directive
- The routines COLSEQ, WTSET, and CSOWN

The intrinsic functions LGE, LGT, LLE, and LLT always use the ASCII collating sequence regardless of the collating sequence you specify.

COLSEQ

The COLSEQ call selects a processor-defined weight table. This call has the form:

CALL COLSEQ (cexp)

cexp

Character expression whose value when any trailing blanks are removed is one of the following (lowercase characters are equivalent):

ASCII

selects the full 256-character standard ASCII collating sequence (default)

ASCII6

selects the OSV$ASCII6_FOLDED collating sequence

ASCII6S

selects the OSV$ASCII6_STRICT collating sequence

COBOL6

selects the OSV$COBOL6_FOLDED collating sequence

COBOL6S

selects the OSV$COBOL6_STRICT collating sequence

DISPLAY

selects the OSV$DISPLAY64_FOLDED collating sequence

DISPLAYS

selects the OSV$DISPLAY64_STRICT collating sequence

DISPLAY63

selects the OSV$DISPLAY63_FOLDED collating sequence

DISPLAY63S

selects the OSV$DISPLAY63_STRICT collating sequence

EBCDIC

selects the OSV$EBCDIC collating sequence

EBCDIC6

selects the OSV$EBCDIC6_FOLDED collating sequence

EBCDIC6S

selects the OSV$EBCDIC6_STRICT collating sequence

INSTALL

selects the OSV$COBOL6_FOLDED collating sequence

The INSTALL option selects the same collating sequence as COBOL6. The weight tables for the above collating sequences are given in appendix C.

The following example selects a collating sequence identical to COBOL6, except that characters $ and . sort equally ('$' .EQ. '.'):

`CALL COLSEQ ('COBOL6')` `CALL WTSET ('$', ICHAR('.'))`	The COBOL standard collating sequence is selected, and the entry for the character code $ (24 hex) is reset to 12 (the value of the weight table indexed by 2E hex ('.')). (ICHAR is an intrinsic function that returns the collating weight of the specified character for the collating sequence currently in effect.)

WTSET

The WTSET call modifies the weight table specified in the COLSEQ call. The WTSET call has the form:

CALL WTSET (ind, wt)

ind

Character expression of length 1.

wt

Integer expression with a value in the range 0 through 255.

If **ind** is a character expression with value c, the element of the weight table indexed by the character code of c is replaced with wt.

CSOWN

The CSOWN call defines a partial collating sequence. This call has the form:

CALL CSOWN (str)

str

Character expression whose value is a sequence of 1 through 256 characters for which a weight table is to be defined. For a string

c(1)c(2)...c(n)

no c(i) can equal c(j) unless i equals j.

CSOWN explicitly defines the weight table elements for the specified characters and then sets all other elements to zero. For i from 1 through n, the weight of c(i) is set to i−1.

The following example illustrates the creation of a user-defined collating sequence:

```
      CHARACTER CTAB*4
      CTAB(1:1)='Z'
      CTAB(2:2)=CHAR(9)
      CTAB(3:3)='$'
      CTAB(4:4)='#'
C$    COLLATE (USER)
      CALL CSOWN (CTAB)
```

The characters to be in the collating sequence are stored in the character variable CTAB. Note that the CHAR function is used to convert the element 9 to an ASCII character. (The character corresponding to the value 9 is the horizontal tab character, which does not have a graphic representation.)

The C$ COLLATE (USER) directive must be specified before the CSOWN call; otherwise, the CSOWN call is ignored. Also, you must specify DEFAULT_COLLATION=USER on the VECTOR_FORTRAN command for a program containing a CSOWN call.

The CSOWN call establishes the new collating sequence, in which Z has weight 0, the horizontal tab has weight 1, $ has weight 2, and # has weight 3. All other characters have weight 0.

Date and Time Information Subprograms

The following subprograms provide date and time information.

DATE

The DATE function returns the current date as the value of the function. The DATE function reference has the form:

DATE ( )

The value returned is in the form yyyy-mm-dd, where yyyy is the year, mm is the number of the month, and dd is the day within the month. The value returned is type character with a length of 10. DATE must be declared type CHARACTER*10 in the calling program.

JDATE

The JDATE function returns the current Julian date as the value of the function. The JDATE function reference has the form:

JDATE ( )

The value returned has the form yyddd, where yy is the year and ddd is the number of the day within the year. The value returned is type character with a length of 5. JDATE must be declared type CHARACTER*5 in the calling program.

TIME or CLOCK

The TIME function or CLOCK function returns the current reading of the system clock as the value of the function. These functions have the forms:

TIME ( )

CLOCK ( )

The value returned is in the form hh:mm:ss, where hh is hours from 0 through 23, mm is minutes, and ss is seconds. The value returned is type character with a length of 8. TIME and CLOCK must be declared type CHARACTER*8 in the calling program.

SECOND

The SECOND function returns the central processor time in seconds that has elapsed since the beginning of the task. This time does not include monitor time, which is operating system overhead time. The SECOND function reference has the form:

SECOND ( )

There is no argument, and the result is real.

Miscellaneous Utility Subprograms

The following subprograms provide for passing information between a user and the operating system.

DISPLA

The DISPLA call places a name and a value in the job log file. This call has the form:

CALL DISPLA (h, k)

h

Character expression to be displayed. It should not exceed 256 characters.

k

Real or integer expression whose value is to be displayed.

The character constant displayed cannot be more than 256 characters; k is a real or integer variable or expression and is displayed as an integer or real value.

REMARK

The REMARK call places a message in the job log file. This call has the form:

CALL REMARK (h)

h

Character expression to be placed in the job log file.

The maximum message length is 256 characters.

SSWTCH

The SSWTCH call tests the system sense switches. This call has the form:

CALL SSWTCH (i, j)

i

Integer expression whose value is a sense switch number. The value must be 1 through 6.

j

Integer variable or array element to receive a value indicating the setting of the sense switch:

1 Sense switch i is on.
2 Sense switch i is off.

If i is out of range, an informative diagnostic is printed, and j is set to 2. The sense switches are set or reset by operations personnel or by the SET_SENSE_SWITCH command.

EXIT

The EXIT call terminates program execution and returns control to the operating system. This call has the form:

CALL EXIT

NOTE

Use of the STOP statement is preferable to CALL EXIT.

CHGUCF

The CHGUCF allows you to disable or enable certain system conditions that terminate program execution from within your FORTRAN program. This call has the form:

CALL CHGUCF (flag, mode, scope, rtncode)

flag

Character expression specifying the name of the NOS/VE User Mask Register condition flag to be changed. Options are:

'DIVIDE_FAULT' or 'DF'
'ARITHMETIC_OVERFLOW' or 'AO'
'EXPONENT_OVERFLOW' or 'EO'
'EXPONENT_UNDERFLOW' or 'EU'
'FP_LOSS_OF_SIGNIFICANCE' or 'FPLOS' or 'FLOS' or 'FP_SIGNIFICANCE'
'FP_INDEFINITE' or 'FPI' or 'FI'
'ARITHMETIC_LOSS_OF_SIGNIFICANCE' or 'ALOS'
or'ARITHMETIC_SIGNIFICANCE'

mode

Logical expression specifying the mode for the specified condition flag (the flag parameter). Options are:

.TRUE.

The condition is enabled (masked ON) for traps.

.FALSE.

The condition is disabled (masked OFF) for traps.

scope

Character expression specifying the scope of reference to be covered by the user mask register setting. Options are:

'LOCAL' or 'L'

The specified flag is changed for the procedure (program or subprogram) that called CHGUCF, and all subprograms called by that program or subprogram.

'GLOBAL' or 'G'

The specified flag is changed for all procedures on the stack at the time of the call to CHGUCF. Any subprograms referenced by the main program, and the main program, are included.

Miscellaneous Utility Subprograms

The following subprograms provide for passing information between a user and the operating system.

DISPLA

The DISPLA call places a name and a value in the job log file. This call has the form:

CALL DISPLA (h, k)

h

Character expression to be displayed. It should not exceed 256 characters.

k

Real or integer expression whose value is to be displayed.

The character constant displayed cannot be more than 256 characters; k is a real or integer variable or expression and is displayed as an integer or real value.

REMARK

The REMARK call places a message in the job log file. This call has the form:

CALL REMARK (h)

h

Character expression to be placed in the job log file.

The maximum message length is 256 characters.

SSWTCH

The SSWTCH call tests the system sense switches. This call has the form:

CALL SSWTCH (i, j)

i

Integer expression whose value is a sense switch number. The value must be 1 through 6.

j

Integer variable or array element to receive a value indicating the setting of the sense switch:

1 Sense switch i is on.
2 Sense switch i is off.

If i is out of range, an informative diagnostic is printed, and j is set to 2. The sense switches are set or reset by operations personnel or by the SET_SENSE_SWITCH command.

EXIT

The EXIT call terminates program execution and returns control to the operating system. This call has the form:

CALL EXIT

NOTE

Use of the STOP statement is preferable to CALL EXIT.

CHGUCF

The CHGUCF allows you to disable or enable certain system conditions that terminate program execution from within your FORTRAN program. This call has the form:

CALL CHGUCF (flag, mode, scope, rtncode)

flag

Character expression specifying the name of the NOS/VE User Mask Register condition flag to be changed. Options are:

'DIVIDE_FAULT' or 'DF'
'ARITHMETIC_OVERFLOW' or 'AO'
'EXPONENT_OVERFLOW' or 'EO'
'EXPONENT_UNDERFLOW' or 'EU'
'FP_LOSS_OF_SIGNIFICANCE' or 'FPLOS' or 'FLOS' or 'FP_SIGNIFICANCE'
'FP_INDEFINITE' or 'FPI' or 'FI'
'ARITHMETIC_LOSS_OF_SIGNIFICANCE' or 'ALOS'
or'ARITHMETIC_SIGNIFICANCE'

mode

Logical expression specifying the mode for the specified condition flag (the flag parameter). Options are:

.TRUE.

The condition is enabled (masked ON) for traps.

.FALSE.

The condition is disabled (masked OFF) for traps.

scope

Character expression specifying the scope of reference to be covered by the user mask register setting. Options are:

'LOCAL' or 'L'

The specified flag is changed for the procedure (program or subprogram) that called CHGUCF, and all subprograms called by that program or subprogram.

'GLOBAL' or 'G'

The specified flag is changed for all procedures on the stack at the time of the call to CHGUCF. Any subprograms referenced by the main program, and the main program, are included.

'PREVIOUS' or 'P'

The specified flag is changed for both the caller of CHGUCF and any other subprograms it calls, and the previous caller (the caller of the caller of CHGUCF) and any other subprograms it calls.

rtncode

Integer variable to receive the resulting return code from the call. The returned value is one of the following values:

<0	Call completed abnormally. An unknown flag name or scope was passed from the caller.
=0	Call completed normally. If the flag was to be turned on, no previous traps for that condition were outstanding.
>0	Call completed normally. If the flag was to be turned on, a previous trap for that condition was outstanding and was cleared before the flag was turned on.

Both uppercase letters and lowercase letters are allowed in any string value used for flag or scope.

The CHGUCF subprogram changes the specified mask flag (condition bit) to the specified mode in the user mask register for all stack frames within the specified scope. The scope applies both for the setting of the user mask register and for the clearing of any associated outstanding trap condition in the user condition register before a condition is to be enabled (turned on). The scope parameter applies to the current and previous calling routines. The setting of the user mask register for a routine holds for that routine and also for any other routines that it calls with subsequent calls or nested calls unless one of those other procedures calls CHGUCF.

Mask and condition register bits not specified are left unchanged. For details on the functioning of traps, the user mask register and user condition register, see the Virtual State Hardware Reference Manual.

It is a good practice to reset the user mask register bits to their system default values and clear any outstanding interrupt conditions in the user condition register before referencing a FORTRAN supplied mathematical intrinsic function.

You can also use the SET_PROGRAM_ATTRIBUTE command to disable or enable these system conditions from outside your program.

Example:

```
      PROGRAM CHECK_MATH
      CHARACTER MESSAGE*80
      INTEGER DF
      DATA X/1.0/,Y/0.0/,DF/1/
C
      MESSAGE = ' '
      CALL CHGUCF('DIVIDE_FAULT',.FALSE.,'LOCAL',DF)
      IF (DF.EQ.0) THEN
          PRINT*, ' CHGUCF CALL WAS CORRECT.'
      ELSE
          PRINT*, ' CHGUCF CALL WAS NOT CORRECT.'
      ENDIF
      Z=X/Y
      PRINT*, BOOL(Z)
C
      CALL CHGUCF('DIVIDE_FAULT',.TRUE.,'LOCAL',DF)
      IF (DF.GT.0) THEN
          MESSAGE = ' DIVIDE FAULT WAS CORRECT.'
      ELSE
          MESSAGE = ' DIVIDE FAULT WAS NOT CORRECT.'
      ENDIF
      END
```

The first call to CHGUCF prevents divide fault errors from terminating program execution in the subsequent assignment statement. The second call to CHGUCF restores the detection of divide faults to cause an execution error.

DEFAUA

The DEFAUA call identifies the location and size of an array that stores the data to be recorded on the AVC$APPLICATION_UNITS statistic (AV11). The DEFAUA call has the form:

CALL DEFAUA (aparray, arraysz, fstat)

aparray

Eight-byte integer array of size 1 to 63. All elements in the array must be positive. Each element represents an event to be counted while the program unit is executing, such as a call to a particular function.

arraysz

Eight-byte integer expression specifying the size of aparray. It must be within the range 1 to 63.

fstat

Character variable to receive the status resulting from the CALL DEFAUA statement; **fstat** must be declared with a length of 264 in the calling program. Use the NOS/VE Status subprograms to retrieve data from this variable.

The DEFAUA call must be in the program unit for which application units are recorded. You should initialize aparray and update each element with the number of units measured by the application before the call to DEFAUA. After program execution, the operating system emits an application units statistic.

See the CYBIL System Interface and the NOS/VE Accounting Analysis System manuals for more information.

Input/Output-Related Subprograms

These subprograms perform operations closely related to input/output, and are described in detail in chapter 7, Input/Output. The input/output-related subprograms are:

- I/O status routines:

  EOF
  IOCHEC
  LENGTH
  LENGTHB
  LENGTHX
  UNIT

- Mass storage I/O routines:

  CLOSMS
  OPENMS
  READMS
  STINDX
  WRITMS

- Miscellaneous routines:

  CONNEC
  DISCON
  IGNFDM
  MOVLEV
  MOVLCH

C$ Directives 11

Listing Control 11-2

Conditional Compilation 11-4

Loader Control 11-6
- Extensible Common 11-6
- Segment Access File Common Blocks 11-7

Collation Control 11-8

DO Loop Control 11-9

External Control 11-10

Vectorization Control 11-11
- C$ ASSUME(NORECUR) Directive 11-11
- C$ ASSUME(MAXITER=c) Directive 11-11
- C$ CONTROL(p) Directive 11-12

C$ PARAM Directive 11-13

Preprocessor Control 11-15

C$ Directives 11

A C$ directive is a special form of comment line that controls compiler processing. A particular C$ directive affects an aspect of the compiler's interpretation of those lines following the directive and preceding either a subsequent directive modifying the same aspect, if such a directive appears, or the end of the program unit. The aspects of interpretation that you can control are:

- Listing of the program and associated compiler-produced information, called listing control
- Specification of program lines to be processed or ignored, called conditional compilation
- Character data comparison collation table, called collation control
- Minimum trip count for DO loops, called DO loop control
- Specification of extensible common or segment access files, called loader control
- Definition of external procedures to be used within your FORTRAN program, called external control
- Definition of SCL parameters to be passed through the execution command (C$ PARAM directive)
- Vectorization, called vectorization control
- Preprocessor control, used to distinguish IM/DM commands from your program

A C$ directive line is identified by the letter C in position 1 together with the character $ in position 2. Such a line is interpreted as a comment if you do not specify COMPILATION_DIRECTIVES=ON on the VECTOR_FORTRAN command. The entire directive must appear on a single line. A C$ directive interrupts statement continuation.

A C$ directive containing a syntax error generally results in a fatal compilation diagnostic.

Listing Control

The listing control directive controls the compiler output list options. This directive has the form

C$ **LIST(p** =*c*,...,*p*=*c*)

p

One of the symbols S, O, R, A, M, or ALL.

c

Optional integer constant:

1 Enable the specified option.

0 Disable the specified option.

If =c is omitted, the effect is the same as p=1.

The listing control directive modifies the state of any initially enabled list option switches. A list option switch is initially enabled when the corresponding list option is requested by the LIST_OPTION parameter on the VECTOR_FORTRAN command. Any attempt to modify a list option switch that was not initially enabled is ignored. A specification of p=0 disables switch p; p=1 enables switch p.

ALL=c is equivalent to S=c, O=c, R=c, A=c, M=c.

A listing control parameter with values other than 0 or 1 results in a warning diagnostic.

The list option switches provide the following control:

S Source lines are listed when enabled.

O Generated object code is listed for statements processed when enabled.

R Symbol references are accumulated for the cross-reference list when enabled. Symbols with no accumulated references do not appear in that list; no accumulation for an entire program unit suppresses cross-reference list.

A The symbol attribute list is generated if this switch is enabled when the END statement is processed.

M The symbol attribute list, DO loop, and common/equivalence map lists are generated if this switch is enabled when the program END statement is processed.

Following example illustrates the listing control directives. All source statements appearing between C$ LIST (S=0) and C$ LIST (S=1) are suppressed in the output listing. (Source statement lines with errors are listed on the file $ERRORS along with diagnostics.) The C$ LIST (ALL=0) directive, active when the END statement is encountered, suppresses the reference map.

```
      PROGRAM P
C     PROGRAM TO TEST LISTING CONTROL DIRECTIVES.

C$    LIST(S=0)
      DIMENSION A(10)

C     THE FOLLOWING CARD CONTAINS A SYNTAX ERROR
C     THE ERROR MESSAGE WILL BE LISTED ON THE $ERRORS FILE.
      INTEGER B/C

C$    LIST(S=1)
      DO 100 I=1,10
      A(I) = 0.0
  100 CONTINUE
C$    LIST(ALL=0)
      END
```

Conditional Compilation

A conditional compilation directive controls whether the lines immediately following the directive are to be processed or ignored by the compiler.

The conditional compilation directives are divided into three categories:

- An IF directive with keyword IF
- An ELSE directive with keyword ELSE
- An ENDIF directive with keyword ENDIF

The IF, ELSE, and ENDIF directives have the following forms:

C$　　**IF(lexp)**, *lab*

C$　　**ELSE**, *lab*

C$　　**ENDIF**, *lab*

lexp

Extended logical constant expression. If a symbolic constant appears, it must have been previously defined in a PARAMETER statement in the program containing the IF directive.

lab

Optional label.

For each IF directive there must appear exactly one ENDIF directive later in the same program unit; likewise, for each ENDIF directive there must appear exactly one IF directive earlier in the same program unit. The lines between an IF directive and its corresponding ENDIF directive are called a conditional sequence. A conditional sequence can optionally contain one ELSE directive corresponding to the IF directive and ENDIF directive delimiting the conditional sequence. An ELSE directive can appear only within a conditional sequence. A conditional sequence cannot contain more than one ELSE directive unless it contains another conditional sequence. If an ELSE directive is contained within more than one conditional sequence, the ELSE directive corresponds to that IF-ENDIF pair which delimits the smallest, that is, innermost, conditional sequence containing the ELSE directive.

If corresponding IF, ELSE, and ENDIF directives have a label, it must be the same label. No other restriction applies to labels on conditional directives. There is no requirement that any conditional directive have a label. The same label can be used on more than one sequence of corresponding conditional directives in a single program unit, including the case of conditional directives whose conditional sequence contains other conditional directives with the same label.

A conditional sequence can contain any number of properly corresponding conditional directives, and therefore other conditional sequences. If two conditional sequences contain the same line, one conditional sequence must lie wholly within the other conditional sequence.

If an IF directive is processed by the compiler and the logical expression has the value true, following lines are processed as if the IF directive had not appeared, unless a corresponding ELSE directive is encountered. In this case, lines between the ELSE directive and the corresponding ENDIF directive are ignored by the compiler. If an IF directive is processed by the compiler and the logical expression has the value false, the following lines are ignored until the corresponding ENDIF directive is encountered, unless a corresponding ELSE directive is encountered. In this case, lines between the ELSE directive and the corresponding ENDIF directive are processed.

The following example illustrates conditional compilation directives. The sample program contains two DO loops. Conditional compilation directives are included to test the value of the symbolic constant M. If M is 1, the first loop is compiled and the second loop is ignored. If M is not 1, the first loop is ignored and the second loop is compiled. Note, however, that the PARAMETER statement sets M = 1.

```
      PROGRAM B
      PARAMETER (M=1)
      DIMENSION A(10)
      DATA A/10*0.0/

C$    IF(M .EQ. 1)
      DO 8 I=1,10
      A(I) = A(I) + 1.0
    8 CONTINUE

C$    ELSE
      DO 12 I=1,10
      A(I) = A(I) - 1.0
   12 CONTINUE

C$    ENDIF

      PRINT*, 'A= ', A
      END
```

Loader Control

The loader control directives cause a named common block to be extensible (using C$ EXTEND) or to associate a named common block with a segment access file (using C$ SEGFILE).

Extensible Common

The C$ EXTEND directive causes a named common block to be extensible. A common block that is extensible has no upper bound other than the size of the memory segment it is contained in. This directive has the form:

C$ EXTEND (/bname/,...,/*bname*/)

bname

A common block name. The name must be defined by a named COMMON statement in the same program unit.

You must declare an extensible common block to be extensible in all program units that define the named common block. For more information about named common, see the COMMON statement description in chapter 4, Specification Statements. Blank common blocks are always extensible.

If the C$ EXTEND directive is used in program units where over-indexing of arrays may occur, then subscript bounds checking should be deactivated with the RUNTIME_CHECKS = NONE option on the VECTOR_FORTRAN command.

The following example shows the use of the C$ EXTEND directive to make the named common blocks (a, b, and c) extensible.

```
      PROGRAM E
      COMMON /a/ a(1), /b/ b(1), /c/ c(1)
C$    EXTEND (/a/,/b/,/c/)
        :
      a(2)=7
      b(2)=8
      END
```

If the program is compiled with the C$ directive ignored (CD=OFF on the VECTOR_FORTRAN command), the assignments 'a(2)=7' and 'b(2)=8' overwrite the original values in the common block for b(1) and c(1).

Compiling the program with the C$ EXTEND (CD=ON on the VECTOR_FORTRAN command) however, protects any over-indexed array references within the bounds of the memory segment.

In the following example, note that only the last dimension of the last array in a common block with more than one entity can be over indexed:

```
      PROGRAM X
      COMMON /A/ a(2), b(3), c(3,4)
C$    EXTEND (/A/)
        ⋮
```

The array element c(3,5) can be referenced while the array element a(3) can not because a(3) would reference the memory location as b(1).

Segment Access File Common Blocks

The C$ SEGFILE loader control directive associates a named common block to a segment access file. After a named common block is associated with a segment access file, you can access the file directly through the common block's variables and arrays. This directive has the form:

C$ SEGFILE (/bname/,..., */bname/***)**

bname

A common block name. The name must be defined by a named COMMON statement in the same program unit.

You must associate, or map, a common block to a segment access file with the OPEN statement by specifying /bname/ for UNIT.

You must not reference a segment access file, or assign values to the associated common block, until after the OPEN statement has opened and associated the file with a common block.

The following example shows the use of the C$ SEGFILE directive to map the common CB1 to the segment access file SFILE:

```
      COMMON /CB1/A(1000)
C$    SEGFILE (/CB1/)
      OPEN (UNIT=/CB1/, FILE='SFILE')
```

Collation Control

The collation control directive specifies whether collation of character relational expressions is directed by the fixed or user-specified weight table. This directive has the form:

C$ COLLATE(p)

p One of the following:

FIXED

Collate according to the fixed (ASCII) weight table.

USER

Collate according to the user-specified (DISPLAY) weight table.

The default is FIXED unless the DEFAULT_COLLATION parameter on the VECTOR_FORTRAN command specifies USER.

A collation control directive directs the interpretation of character relational expressions and of CHAR or ICHAR intrinsic function references in the lines following the directive and preceding either another collation control directive or the END statement of the program unit. In the case of a character relational expression or a CHAR or ICHAR reference in a statement function definition, the collation that applies is that in effect for the line or lines containing a reference to the statement function. The following example shows a character relational expression used in a statement function:

```
      PROGRAM P
      LOGICAL LSF
      CHARACTER*5, X, Y, S, T
C$    COLLATE(USER)
      LSF(X,Y) = X.LT.Y
C$       :
      COLLATE(FIXED)
      IF (LSF(S,T)) A=1.0
         :
      END
```

The reference LSF(S,T) results in an evaluation of the character relational expression S.LT.T according to the fixed weight table.

DO Loop Control

The DO loop control directive controls the minimum trip count for DO loops. This directive has the form:

C$ **DO (OT=*c*)**

c

Integer constant or integer symbolic constant:

0 Minimum trip count is zero

1 Minimum trip count is one

If =c is omitted, minimum trip count is zero.

The DO loop control directive modifies the state of the DO loop switch. The DO loop switch is initially set according to the presence or absence of the ONE_TRIP_DO parameter on the VECTOR_ FORTRAN command. A DO loop control directive overrides the ONE_TRIP_DO request.

The DO loop directive controls the minimum trip count for all loops that follow the directive, until either an END statement or another DO loop directive that resets the switch is encountered.

A DO loop control directive affects the interpretation of only those DO loops whose DO statements follow the directive in the same program unit.

External Control

The external control directive allows a FORTRAN program to recognize and call a routine written in a language with different naming conventions and calling sequences than FORTRAN. The directive has the form:

C$ EXTERNAL(ALIAS='exname', LANG=lspec), name

exname

Name of the external routine; can be up to 31 characters.

lspec

Selects the programming language in which the external procedure is written. Options are:

C

Selects the C programming language

CYBIL

Selects the CYBIL programming language

name

Name of the routine as it is to be known in your FORTRAN program. Must be a valid FORTRAN program unit name.

The C$ EXTERNAL allows you to use a FORTRAN name to call a routine written in other languages. For more information on calling routines from a FORTRAN program, see Calling Other Language Subprograms in appendix D. The following example shows a FORTRAN program that calls a C routine named c_program:

```
      PROGRAM M
         :
C$    EXTERNAL(ALIAS='c_program', LANG=C), CPROG
      CALL CPROG(JCOUNT)
      END
```

Vectorization Control

The vectorization control directives are used to control the vectorization of lines in your program. There are three types of vectorization control directives:

C$ ASSUME(NORECUR)
C$ ASSUME(MAXITER=c)
C$ CONTROL(p)

For more information about vectorization and the vectorization control directives, see Controlling Implicit Vectorization with C$ Directives in chapter 13, Vectorization.

C$ ASSUME(NORECUR) Directive

The C$ ASSUME(NORECUR) directive indicates that no recurrence cycle exists in the immediate following statement. This directive has the form:

C$ ASSUME(NORECUR)

The C$ ASSUME(NORECUR) directive can only appear within a DO loop. For example:

```
      DO 5 I = 1, N
C$    ASSUME(NORECUR)
      A(I) = A(I + K) + 1.0
    5 CONTINUE
```

The C$ ASSUME directive eliminates the possibility of the variable K causing a recurrence cycle.

C$ ASSUME(MAXITER=c) Directive

The C$ ASSUME directive indicates the maximum iteration count of a DO loop. The C$ ASSUME directive has the form:

C$ ASSUME(MAXITER=c)

where c is an integer constant or symbolic constant.

The C$ ASSUME(MAXITER=c) directive must appear directly before a DO statement; otherwise it is ignored.

Example:

```
C$    ASSUME(MAXITER = 100)
      DO 5 I = 1, M + N
         A(I) = B(I) + 1.0
    5 CONTINUE
```

The C$ ASSUME directive eliminates the possibility that the sum of M and N exceeds 100.

C$ CONTROL(p) Directive

The C$ CONTROL directive indicates which source lines of a program are ignored in vectorization. The C$ CONTROL directive has the form:

C$ CONTROL(p)

where

p One of the following:

VEC

Enables vectorization for subsequent lines. The compiler attempts to vectorize all subsequent lines until a C$ CONTROL(NOVEC) directive appears or the end of the program unit is encountered.

NOVEC

Disables vectorization for subsequent lines.

Example:

```
      DO 5 I = 1, N
      A(I) = B(I) + C(I)
C$    CONTROL(NOVEC)               ! <--- Vectorization
       DO 7 J = 1, M               !   -- is disabled
        R(I,J) = 2.0 * S(I,J)      !   -- for these
    7  CONTINUE                    !   -- lines
C$    CONTROL(VEC)                 ! <--- Vectorization
      D(I) = A(I) - 1.0            !   -- is enabled for
    5 CONTINUE                     !   -- these and
                                   !   -- following lines
```

The inner DO loop is not a candidate for vectorization. The first and last statements of the outer DO loop are candidates for vectorization.

If a DO statement is written within the range of a C$ CONTROL(NOVEC) directive, the control variable for the DO loop will not be sectioned even if a subsequent C$ CONTROL (VEC) directive appears within the DO loop. Any inner loops nested within the DO loop will be candidates for vectorization if vectorization has been enabled by the C$ CONTROL(VEC) directive. Example:

```
C$    CONTROL(NOVEC)
      DO I=1, 10
C$    CONTROL(VEC)
      DO J=1, 10
      A(I,J) = B(I, J) + C(I,J)
   10 CONTINUE
```

Vectorization is disabled for index I in the inner DO loop and enabled for index J.

C$ PARAM Directive

The C$ PARAM directive defines parameters that are to be specified on the execution command. The general form of the C$ PARAM directive is

C$ PARAM(pdefs)

where pdefs is a character constant expression, whose value is of the form 'pdef; ...; pdef', where pdef is a valid SCL parameter definition. The format for parameter definitions is described in the NOS/VE System Usage manual.

The characters C$ must appear in positions 1 and 2, and the string PARAM must begin in or after position 7. The C$ PARAM directive cannot be continued on a subsequent line. Long parameter definitions can be made by specifying the required character constant in one or more PARAMETER statements, each of which can be continued over 19 lines, and then referencing the symbolic name of the constant in the C$ PARAM directive.

Any parameter that appears on the execution command must have been defined by a C$ PARAM directive in the program to be executed. The C$ PARAM directive defines such parameter properties as:

- Parameter name
- Whether or not the parameter is required on the execution command
- Parameter default values
- Parameter kind
- Allowable number of value sets
- Allowable number of value elements in a value set

A parameter specified on the execution command must conform to its definition in the C$ PARAM directive.

Only one C$ PARAM directive can be specified in a program, and it must appear in the main program. It can be placed anywhere after the PROGRAM statement, and can define one or more parameters. A C$ PARAM directive in a subprogram has no effect.

If a program contains a C$ PARAM directive, the PROGRAM statement should contain only the program name; no file equivalencing can be done.

Example:

```
C$    PARAM('A:INTEGER; B:VAR OF STRING')
```

This directive defines two parameters. Values specified for parameter A will be of kind integer; and values specified for parameter B will be SCL variables of kind string.

The following example shows how to represent five parameters with a single C$PARAM directive:

```
        ⋮
      CHARACTER*(*) P1, P2, P3, P4, P5
      PARAMETER(P1 = 'P1:BOOLEAN;')
      PARAMETER(P2 = 'P2:LIST 1 .. 7, 2 .. 2 RANGE OF INTEGER -3 .. 10;')
      PARAMETER(P3 = 'P3:RANGE OF VAR OF STATUS;')
      PARAMETER(P4 = 'P4:LIST 2 .. 4 OF FILE;')
      PARAMETER(P5 = 'P5:RANGE OF INTEGER')
C$    PARAM (P1//P2//P3//P4//P5)
        ⋮
```

Preprocessor Control

A preprocessor is a program that prepares data for further processing by arranging it into different formats.

The IM/DM preprocessor, or precompiler, scans your FORTRAN program and expands IM/DM commands into one or more FORTRAN statements. The IM/DM preprocessor produces preprocessor control directives to distinguish the expanded statements from the statements you write.

NOTE

Unlike the other C$ directives described in this chapter, you do not enter the preprocessor directives in your FORTRAN program. They are generated by the IM/DM preprocessor. They are described here to assist you in viewing a listing of a program that has been processed by the preprocessor.

When viewing the output generated by the IM/DM preprocessor, the expanded lines are delimited by the preprocessor control directives. The preprocessor control directives are:

C$ PP(PS)

Indicates the start of the preprocessor commands. The preprocessor commands immediately following are to be listed and line numbers assigned.

C$ PP(PE)

Indicates the start of the expanded source resulting from the preprocessor commands. The expanded source lines are suppressed on the source listing.

C$ PP(EE)

Indicates the end of the expanded source resulting from the preprocessor commands.

C$ PP(ERR, SEV=m, ID=id, C=nnnnnn, *MTM=boolean*)

Indicates the start of a preprocessor error message where

m

The severity of the message level. One of the following values: I (Informational), W (Warning), F (Fatal), or C (Catastrophic).

id

The ID field of the error.

nnnnnn

The CONDITON field of the error.

If the MTM parameter specifies YES, or is not included in the C$ PP statement, the lines up to the C$ PP(ERREND) statement are the message text of the error.

C$ PP(ERREND)

Indicates the end of a preprocessor error message.

Preprocessor control directives can be nested, similar to a nested DO loop or If block.

The lines delimited by C$ PP(PE) and C$ PP(EE) do not appear in the source listing produced by the compiler.

NOTE

You should not place preprocessed programs in Source Code Utility (SCU) libraries.

Compilation and Execution 12

VECTOR_FORTRAN Command 12-1
Parameter Overview 12-2
Parameter Formats 12-5
Parameter Names 12-7
File Positioning 12-7
Parameter Options 12-8
BINARY_OBJECT (B, BO, BINARY) 12-8
COMPILATION_DIRECTIVES (CD) 12-8
DEBUG_AIDS (DA) 12-8
DEFAULT_COLLATION (DC) 12-9
ERROR (E) 12-9
ERROR_LEVEL (EL) 12-10
EXPRESSION_EVALUATION (EE) 12-10
FORCED_SAVE (FS) 12-11
INPUT (I) 12-12
INPUT_SOURCE_MAP (ISM) 12-12
LIST (L) 12-12
LIST_OPTIONS (LO) 12-13
MACHINE_DEPENDENT (MD) 12-14
ONE_TRIP_DO (OTD) 12-15
OPTIMIZATION_LEVEL (OL, OPTIMIZATION, OPT) 12-16
OPTIMIZATION_OPTIONS (OO) 12-17
REPORT_OPTIONS (RO) 12-18
RUNTIME_CHECKS (RC) 12-18
STANDARDS_DIAGNOSTICS (SD) 12-19
STATUS 12-19
TARGET_MAINFRAME (TM) 12-20
TERMINATION_ERROR_LEVEL (TEL) 12-21
VECTORIZATION_LEVEL (VL, VECTORIZATION, VEC) 12-21
VECTOR_FORTRAN Command Examples 12-22

Compiler Output Listing 12-24
Source Listing 12-24
Compilation Statistics 12-24
Reference Map 12-25
General Format of Maps 12-26
Variables Map 12-28
Symbolic Constants Map 12-30
Namelist Map 12-31
Common and Equivalence Map 12-32
Statement Labels Map 12-33
DO Loops Map 12-34
Entry Points Map 12-35
Procedures Map 12-36
Input/Output Units Map 12-38
Unclassified Names Map 12-39

Execution Command 12-40
$PRINT_LIMIT Parameter 12-41
STATUS Parameter 12-42
User-Defined System Command Language Parameters 12-43
File Name Substitution 12-43

Compilation and Execution 12

NOS/VE provides commands for compiling and executing FORTRAN Version 2 programs. The FORTRAN Version 2 compiler reads the input source program and produces an output object program. The object program can be loaded into memory and executed by an execution command.

VECTOR_FORTRAN Command

The FORTRAN Version 2 compiler is called and executed by the VECTOR_FORTRAN command. This command selects a variety of compiler options, including files to be used for input and output, type of output to be produced, and level of vectorization. The VECTOR_FORTRAN command has the form:

VECTOR_FORTRAN or
VECF or
VFORTRAN or
VFTN or
FORTRAN2 or
FTN2
INPUT=list of file
BINARY_OBJECT=file
LIST=file
COMPILATION_DIRECTIVES=boolean
DEBUG_AIDS=list of keyword
DEFAULT_COLLATION=keyword
ERROR=file
ERROR_LEVEL=keyword
EXPRESSION_EVALUATION=list of keyword
FORCED_SAVE=boolean
INPUT_SOURCE_MAP=list of file
LIST_OPTIONS=list of keyword
MACHINE_DEPENDENT=keyword
ONE_TRIP_DO=boolean
OPTIMIZATION_LEVEL=keyword
OPTIMIZATION_OPTIONS=list of keyword
REPORT_OPTIONS=keyword
RUNTIME_CHECKS=list of keyword
STANDARDS_DIAGNOSTICS=keyword
TARGET_MAINFRAME=keyword
TERMINATION_ERROR_LEVEL=keyword
VECTORIZATION_LEVEL=list of keyword
STATUS=status variable

The terms file, keyword, boolean, and status variable are defined in the NOS/VE System Usage manual.

Parameter Overview

Parameters on the VECTOR_FORTRAN command select the desired options. The parameters can be specified either by name or positionally (except the STATUS parameter, which must be specified by name), and must be separated by a comma or one or more blanks. If a comma is used, it can be followed by one or more blanks. Parameters specified by name can appear in any order. The required order for parameters specified positionally is shown in the preceding format description.

When you specify a parameter positionally, you must indicate the position of any omitted parameters that precede the specified parameter by commas; the first omitted parameter is indicated by two successive commas, and each additional omitted parameter is indicated by an additional comma. Thus, if *n* parameters are omitted, *n+1* commas are required. For example, the following commands are equivalent:

```
/vector_fortran input=sfile optimization_level=high

/vector_fortran sfile, , , , , , , , , , , , , , ,high
```

The second command specifies the INPUT and OPTIMIZATION_LEVEL parameters positionally; the positions of the omitted parameters are indicated by fourteen successive commas.

If you omit any parameter from the VECTOR_FORTRAN command, a default is automatically provided. Since this default corresponds to the most commonly used option for the parameter, you need specify only a few parameters or none at all, in most cases. The parameter names, the processor-supplied defaults, and brief descriptions are presented in alphabetical order in table 12-1.

Unrecognizable parameters prevent compilation from beginning. Conflicting options are either resolved by the compiler or prevent compilation from beginning, depending on the severity of the conflict. Any conflict the compiler resolves is indicated by a job log entry. You should not specify any parameter more than once. If you do so, the operating system issues an error message and the compilation does not begin.

Table 12-1. VECTOR_FORTRAN Command Parameters

Parameter Name	Abbreviated Form	Description	Default
BINARY_OBJECT	B BO BINARY	Binary output file	B=$LOCAL.LGO
COMPILATION_ DIRECTIVES	CD	C$ directive suppression	CD=ON (directives recognized)
DEBUG_AIDS	DA	Debugging options	DA=NONE
DEFAULT_ COLLATION	DC	Collating sequence control	DC=FIXED
ERROR	E	File to receive error information	E=$ERRORS
ERROR_LEVEL	EL	Severity level of error messages to be printed	EL=WARNING
EXPRESSION_ EVALUATION	EE	Order of evaluation of expressions	EE=NONE
FORCED_SAVE	FS	Save variables and arrays in subprograms	FS=OFF (variables and arrays not saved)
INPUT	I	Source input file(s)	I=$INPUT
INPUT_SOURCE_ MAP	ISM	Source map file(s)	ISM=$NULL
LIST	L	Output listing file	L=$LIST
LIST_OPTIONS	LO	Output listing options	LO=S
MACHINE_ DEPENDENT	MD	Machine dependencies	MD=NONE (machine dependencies not flagged)
ONE_TRIP_DO	OTD	Minimum trip count for DO loops	OTD=OFF (zero trip loops)
OPTIMIZATION_ LEVEL	OL OPT OPTIMIZATION	Compiler optimization level	OL=LOW

(Continued)

Table 12-1. VECTOR_FORTRAN Command Parameters *(Continued)*

Parameter Name	Abbreviated Form	Description	Default
OPTIMIZATION_ OPTIONS	OO	Instruction scheduling	OO=NONE
REPORT_OPTIONS	RO	Level of detail on vectorizer report	RO=NONE
RUNTIME_ CHECKS	RC	Runtime range checking	RC=NONE
STANDARDS_ DIAGNOSTICS	SD	Diagnose non-ANSI usages	SD=NONE
STATUS	-	SCL variable to receive error status information	None
TARGET_ MAINFRAME	TM	Vector machine for which code is generated	TM=C180_CURRENT_ MAINFRAME
TERMINATION_ ERROR_LEVEL	TEL	Severity level of errors for which abnormal status returned	TEL=FATAL
VECTORIZATION_ LEVEL	VL VEC VECTORIZATION	Compiler vectorization level	VL=NONE

Parameter Formats

The following paragraphs briefly describe the formats of parameters used on the VECTOR_FORTRAN command. For more information on SCL parameter formats in general, refer to the NOS/VE System Usage manual.

The VECTOR_FORTRAN command parameters have the following general format:

parameter name=value list

There are five general types of parameters used in the VECTOR_FORTRAN command. Four of these types are described below. The fifth type, of which there is only a single parameter (the STATUS parameter), is described under Parameter Options.

- The first type of parameter requires you to specify a file. These parameters have the form:

  parameter name=file

  or

  parameter name=list of file

  A file specification identifies a local or permanent file. A file specification consists of a file path (which includes the file name), an optional cycle reference (for permanent files), and an optional file position. If a list of files is allowed, separate each file name with a comma or space and enclose the list in parentheses. The general form of a file reference is:

  path.*cycle_reference.file_position*

  The cycle reference and file position are optional. The simplest form of a file reference is a file name. Refer to the NOS/VE System Usage manual for more information on file specifications.

  Example:

  ```
  /vector_fortran input=(infile, file_a) binary_object=:fam.use.bin
  ```

  This command specifies the file names INFILE and FILE_A for the INPUT parameter; the binary object file BIN is written to family FAM with user name USE.

- The second type of parameter requires you to specify a keyword corresponding to the desired option. These parameters have the form:

  parameter name=keyword

  Example:

  ```
  /vector_fortran error_level=warning
  ```

  This command specifies the keyword WARNING for the ERROR_LEVEL parameter.

- The third type of parameter has two possible settings: on or off. These parameters have the form:

  parameter name=boolean

  To turn the option on, specify parameter name=ON. To turn the option off, specify parameter name=OFF. An example of this type of parameter is the COMPILATION_DIRECTIVES parameter.

  Example:

  ```
  /vector_fortran compilation_directives=on
  ```

  This command selects the option to process compilation directives.

- The fourth type of parameter allows you to select a list of options. These parameters have the form:

  parameter name=list of keyword

  A particular option is selected by specifying the keyword associated with that option; if the keyword is omitted, the option is not selected. If you select more than one option, each keyword must be separated by a comma or blank and the list of keywords must be enclosed in parentheses. If you select a single option, the parentheses can be omitted. All options for a particular parameter, including the default options, can be deselected by specifying parameter name=NONE.

  Example:

  ```
  /vector_fortran list_options=(a,o,sa) expression_evaluation=none
  ```

  This command selects the A, O, and SA options for the LIST_OPTIONS parameter, and deselects all options for the EXPRESSION_EVALUATION parameter.

Parameter Names

Each parameter name (except the STATUS parameter) has a long form and one or more abbreviated forms. In addition, some of the parameter keywords have both a long and an abbreviated form. Both forms have the same meaning and either can be specified. For example, the following commands are equivalent:

```
/vector_fortran b=bin tel=f

/vector_fortran binary_object=bin termination_error_level=fatal
```

The parameter names and associated abbreviations are shown in table 12-1.

File Positioning

The input and output files specified on the VECTOR_FORTRAN command have a default position that can be altered by use of a file position indicator. The file position indicator allows you to specify how a particular file is to be positioned before it is used. The file position indicators are:

$BOI

Positions the file at the beginning-of-information (BOI).

$EOI

Positions the file at the end-of-information (EOI).

$ASIS

Does not position the file (the file is read or written beginning at its current position).

If you omit the file position indicator, the file remains at its current position (or is repositioned according to the OPEN_POSITION file attribute), with the exception that the system file $OUTPUT is positioned at EOI. (Although $OUTPUT is normally positioned at BOI, it is connected to the physical file OUTPUT which is positioned at EOI. As long as this connection remains intact, the effect is to position $OUTPUT at EOI.)

Example:

```
/vector_fortran input=sprog.$asis binary_object=:fam.use.bin.$eoi
```

The source input file SPROG is read beginning at the current position. The binary output file BIN is written to family FAM with user name USE, and is positioned at the end-of-information.

Parameter Options

Following are descriptions of the options for each of the VECTOR_FORTRAN command parameters. The parameters are listed in alphabetical order. The alphabetical order is different from their positional order. The heading that precedes each parameter description consists of the parameter name followed by the abbreviation enclosed in parentheses.

BINARY_OBJECT (B, BO, BINARY)

The BINARY_OBJECT parameter specifies the file to receive the binary object code produced by the compiler. The binary object code will be generated into a single file even if you specified a list of files for the INPUT parameter. Options are:

file

Binary object code is written to the specified file.

$NULL

Binary object code is written to file $NULL. (The object code is discarded.)

If BINARY_OBJECT is omitted, $LOCAL.LGO is used.

COMPILATION_DIRECTIVES (CD)

The COMPILATION_DIRECTIVES parameter controls the recognition of C$ directives within the source program. Options are:

ON

C$ directives are processed.

OFF

C$ directives are not processed. (They are treated as comments.)

If COMPILATION_DIRECTIVES is omitted, ON is used.

DEBUG_AIDS (DA)

The DEBUG_AIDS parameter selects debugging options. Options are:

DT

Generates line number and symbol tables for use by the Debug utility.

PC

Generates code to allow for load-time argument checking. Argument mismatch information is written to the $ERRORS file at load time, regardless of the LOAD_MAP_OPTION on the EXECUTE_TASK or SET_PROGRAM_ATTRIBUTE commands.

ALL

Selects both DEBUG_AIDS=DT and DEBUG_AIDS=PC options.

NONE

No options are selected.

If DEBUG_AIDS is omitted, NONE is used.

NOTE

You can use the ABORT_FILE parameter on the EXECUTE_TASK command to specify a file containing Debug commands to be executed if your program aborts. The Debug commands are used only if the program is not executed in Debug mode (the DEBUG_MODE parameter specifies OFF).

DEFAULT_COLLATION (DC)

The DEFAULT_COLLATION parameter specifies the weight table to be used for the evaluation of character relational expressions and by the CHAR and ICHAR intrinsic functions. Options are:

USER (U)

A user-specified weight table is used.

FIXED (F)

The fixed weight table is used.

If DEFAULT_COLLATION is omitted, FIXED is used.

For more information about weight tables, see Collating Sequence Control Subprograms in chapter 10, NOS/VE and Utility Subprograms.

ERROR (E)

The ERROR parameter specifies the file to receive compiler-generated error information. In the event of an error of ERROR_LEVEL-specified severity or higher, a diagnostic is written to the file specified by the ERROR parameter.

If a listing file (LIST parameter) is also specified, the diagnostic is written to both files.

If ERROR is omitted, error information is written to the file $ERRORS.

ERROR_LEVEL (EL)

The ERROR_LEVEL parameter determines the severity level of errors to be printed on the output listing. Selection of a particular option specifies that level and all higher (more severe) levels. Options are (in order of increasing severity):

INFORMATIONAL (I)

Lists informational, warning, fatal, and catastrophic errors. The syntax of trivial errors is correct but the usage is questionable.

WARNING (W)

Lists warning, fatal, and catastrophic errors. Warning errors are errors where the syntax is incorrect but the compiler has made an assumption (such as adding a comma) and continued processing.

FATAL (F)

Lists fatal and catastrophic errors. Fatal errors prevent the compiler from processing the statement containing the errors. The compiler continues processing after a fatal error; however, executable object code is not generated.

CATASTROPHIC (C)

Lists catastrophic errors. Catastrophic errors terminate compilation of the current program unit. Compilation continues with the next program unit.

If ERROR_LEVEL is omitted, WARNING is used.

EXPRESSION_EVALUATION (EE)

The EXPRESSION_EVALUATION parameter controls the way the compiler evaluates expressions.

When evaluating expressions, the compiler normally performs certain optimizations in order to produce more efficient object code. In most cases, these optimizations have no effect on program execution. However, under certain numerically unstable conditions, these optimizations can alter the results of execution. Such conditions usually arise when a program uses values that approach the limits of the computer, or when an operation such as a multiplication or division combines a very large operand with a very small one.

The EXPRESSION_EVALUATION parameter prevents the compiler from performing optimizations which might affect the results of execution. The EXPRESSION_EVALUATION parameter should be used only when necessary. Options are:

NONE

Causes no options to be selected.

(op, ..., *op*)

where op is one of the following:

CANONICAL (C)

Directs the compiler to evaluate expressions strictly according to the precedence rules described in chapter 5, Expressions and Assignment Statements. If you select this option, the compiler interprets each expression as if parentheses had been used to completely specify the order in which

operations are performed. If you do not select this option, the compiler may reorder operations that are mathematically associative, commutative, or distributive.

MAINTAIN_EXCEPTIONS (ME)

Prevents the compiler from performing optimizations that eliminate instructions that might cause runtime errors. Also causes relational expressions involving real or double precision operands to be evaluated using floating-point comparisons. If not specified, integer (bit-by-bit) comparison is used.

MAINTAIN_PRECISION (MP)

Prevents the compiler from performing optimizations that change a floating-point operation to a form that is mathematically equivalent but not computationally equivalent.

OVERLAPPING_STRING_MOVES (OSM)

Guarantees valid character assignment in character assignment statements of the form *v=exp* where the character positions being defined in *v* are reference in *exp*.

REFERENCE (R)

Causes intrinsic functions to be called by reference rather than by value. Also results in the generation of descriptive error messages by internal FORTRAN routines if execution errors occur. If this option is not selected, the operating system produces error messages which generally provide less information.

If EXPRESSION_EVALUATION is omitted, NONE is used.

You can use the EXPRESSION_EVALUATION parameter to detect numerically unstable conditions such as those described above. If the results obtained with a particular EXPRESSION_EVALUATION option differ significantly from the results obtained without the option, numerical instability exists.

FORCED_SAVE (FS)

The FORCED_SAVE parameter specifies whether or not the values of variables and arrays in subprograms are to be retained after execution of a RETURN or END statement. For subprograms compiled under OL=DEBUG or OL=LOW, however, values are always retained regardless of the FORCED_SAVE option selected. Options are:

ON

Variable and array values are saved after execution of a RETURN or END statement. This option is equivalent to specifying a SAVE statement in every subprogram compiled.

OFF

Variable and array values are not saved after execution of a RETURN or END statement.

If FORCED_SAVE is omitted, OFF is used.

INPUT (I)

The INPUT parameter specifies the file containing the input source code. Options are:

file or list of file

Source code to be compiled is contained in the specified file or files. Each file must contain a full program unit and not parts of a program unit. If more than one file is specified, the list of files is enclosed in parentheses.

If INPUT is omitted, $INPUT is used. There is no default connection for interactive jobs. You can create a file connection by using the NOS/VE CREATE_FILE_CONNECTION command as described in the NOS/VE Commands and Functions manual.

INPUT_SOURCE_MAP (ISM)

The INPUT_SOURCE_MAP parameter is used when the FORTRAN source program is contained in SCU decks. The ISM parameter specifies the file containing the source map that was generated by the OUTPUT_SOURCE_MAP option on the SCU_EXPD command.

If more than one file is specified, the list of files is enclosed in parentheses.

If INPUT_SOURCE_MAP is omitted, $NULL is used.

A program compiled with a source map file, and DA=DT or ALL, can be used in the full screen Debug utility. For more information about SCU decks, see the NOS/VE Source Code Utility manual.

LIST (L)

The LIST parameter specifies the file to receive the compiler output listing. This includes the source listing, diagnostics, compile-time statistics, and information requested by the LIST_OPTIONS parameter.

The compiler output listing is described later in this chapter under Compiler Output Listing. The format of the compiler output listing is as though you had combined the input files into a single file if you specified a list of files for the INPUT parameter.

If LIST is omitted, $LIST is used.

The default connection for $LIST is $NULL to batch jobs; data written to file $NULL is discarded. There is no default connection for interactive jobs. You can create a file connection using the NOS/VE CREATE_FILE_CONNECTION command as described in the NOS/VE Commands and Functions manual.

LIST_OPTIONS (LO)

The LIST_OPTIONS parameter specifies the information that is to appear on the compiler output listing. (This information is described later in this chapter under Compiler Output Listing.) The information is written to the file specified by the LIST parameter. If the LIST parameter is not specified, you must connect the default file by using the NOS/VE command:

```
/create_file_connection  standard_file=$list  file=$user.list_file
```

The LIST_OPTIONS parameter allows you to select multiple options. Options are:

NONE

No output listing is produced.

(op, ..., op)

where op is one of the following:

A

A listing of the attributes of each symbolic name used or defined within the program is produced. These attributes include data type, class, and so forth.

R

A cross reference listing is produced. This listing shows the locations of the definition and use of each symbolic name in the program. Names that are defined but not referenced are not listed.

M

A symbol attribute list (same as A option), DO loop map, and common block map are produced. The DO loop portion lists all DO loops in the program, including implied DO lists, and their properties. The common block portion describes the storage layout for common blocks, and the equivalence-induced storage overlap for all variables and arrays.

S

A listing of the program source statements is written to the output file. Lines turned off by C$ LIST directives are not listed.

SA

Same as S, except that lines turned off by C$ LIST directives are listed.

O

A listing of the generated object code is provided.

If LIST_OPTION is omitted, S is used.

MACHINE_DEPENDENT (MD)

The MACHINE_DEPENDENT parameter specifies whether the use of machine dependent capabilities within the program is to be diagnosed and if so, how severely. These capabilities include coding that depends on the number of characters in a word, such as the boolean data type, ENCODE and DECODE statements, and certain uses of BUFFER IN and BUFFER OUT. Options are:

NONE

Machine dependent usages are not diagnosed.

INFORMATIONAL (I)

Machine dependent usages are diagnosed as informational errors.

WARNING (W)

Warning messages are printed for machine dependent usages.

FATAL (F)

Machine dependent usages are treated as fatal errors, which result in a nonexecutable object program.

If MACHINE_DEPENDENT is omitted, NONE is used.

ONE_TRIP_DO (OTD)

The ONE_TRIP_DO parameter establishes the minimum trip count for DO loops. Options are:

ON

Minimum trip count for DO loops is one.

OFF

Minimum trip count for DO loops is zero.

If ONE_TRIP_DO is omitted, OFF is used.

The trip count is the number of times a DO loop is executed. Specifying ONE_TRIP_DO=ON sets the minimum trip count to one. This means that all DO loops execute at least once, even if the terminating conditions are satisfied before the loop is initially entered. This information enables the compiler to generate more efficient object code.

Specifying ONE_TRIP_DO=OFF sets the minimum trip count to zero; this means that DO loops whose terminating conditions are satisfied before the loop is initially entered will not be executed.

You can override the ONE_TRIP_DO parameter with the C$ DO directive. See DO loops in chapter 6, Flow Control Statements.

If you specify ONE_TRIP_DO=ON and VECTORIZATION_LEVEL=HIGH, be sure that all DO loops in your program have a minimum trip count of one. If a DO loop that is being vectorized has a zero or negative minimum trip count, the vectorized version of the DO loop does not execute once as it does in the scalar version.

For Better Performance

For full optimization of DO loops you should specify ONE_TRIP_DO=ON. You should check all DO loops in your program to ensure that the results will not be affected by executing all DO loops at least once.

OPTIMIZATION_LEVEL (OL, OPTIMIZATION, OPT)

The OPTIMIZATION_LEVEL parameter selects the level of optimization performed by the compiler. Options are:

DEBUG

Object code that is modified for debugging use. Also automatically selects FORCED_SAVE=ON.

LOW

Minimum optimization is performed, resulting in faster compilation time but slower execution time.

HIGH

Maximum optimization is performed, resulting in slower compilation time but faster execution time.

If OPTIMIZATION_LEVEL is omitted, LOW is used.

Selecting OPTIMIZATION_LEVEL=LOW or DEBUG results in object code that can be executed on any model of the CYBER 180.

NOTE

At OPTIMIZATION_LEVEL=HIGH, storage is not allocated at load time for variables and arrays unless they are in a common block, saved (by a SAVE statement or FORCED_SAVE=ON compiler option), initialized in a DATA statement, or used as actual arguments. Instead, storage is allocated for them on the runtime stack during execution, only when the program unit to which they belong becomes active. This storage is then given up on execution of a RETURN or END statement in the program unit.

The default runtime stack size is about 2 million bytes. If this limit is exceeded, a runtime error results, usually of the form "Tried to read/write beyond maximum segment length" or "A stack segment contains invalid frames."

For programs where the number of active items allocated on the runtime stack exceeds the default limit, you can increase the runtime stack size by specifying the STACK_SIZE parameter on the EXECUTE_TASK command (as described in the NOS/VE Object Code Management manual).

For Better Performance

For full optimization (fastest execution), specify OPTIMIZATION_LEVEL=HIGH. However, this option results in slower compilation.

OPTIMIZATION_OPTIONS (OO)

The OPTIMIZATION_OPTIONS parameter specifies certain optimizations in the object code. This parameter is only significant when the OPTIMIZATION_LEVEL parameter specifies HIGH. Options are:

ALTERNATIVE_CODE_SELECTION (ACS)

For object code that is going to be executed on a model 990 or 995, ALTERNATIVE_CODE_SELECTION generates two versions of object code for DO loops whose iteration count is not known at compile time. One version uses vector instructions and the other version uses scalar instructions. The vector version is used at runtime unless the overhead for using vector instructions is larger than the time savings achieved with the vector version. In that case, the scalar version is used.

INSTRUCTION_SCHEDULING (IS)

Selects instruction scheduling. Instruction scheduling allows for improved execution on the Model 990 or 995 regardless of the machine on which compilation occurs.

NONE

Instruction scheduling is determined by the values of the OPTIMIZATION_LEVEL and TARGET_MAINFRAME parameters as follows:

Compilation Machine	TM Parameter	OL Parameter	Instruction Scheduling
990	C180CM (default)	HIGH	YES
990	C18V	HIGH	YES
990	C180MI	HIGH	NO
Non-990	C180CM (default)	HIGH	NO
Non-990	C180V	HIGH	YES
Non-990	C180MI	HIGH	NO

If OPTIMIZATION_OPTIONS is omitted, NONE is used.

For Better Performance

The OPTIMIZATION_OPTIONS parameter can be used to produce code that is optimized for a particular class of machine. If the code is executed on the same machine it is compiled on, then the best code is usually produced by leaving this parameter at the default value. For some programs, this will not be true and experimentation can indicate the best action to take.

REPORT_OPTIONS (RO)

The REPORT_OPTION parameter specifies the level of detail of the vectorizer report messages to appear on the listing file. Options are:

BRIEF (B)

Selects brief mode messages only. Brief mode messages are a subset of full mode messages and describe the results of vectorization on your program.

FULL (F)

Selects full mode messages. Full mode messages describe all aspects of vectorization that take place on your program.

NONE

No vectorizer messages are reported on the listing file.

If REPORT_OPTIONS is omitted, NONE is used.

You must specify VECTORIZATION_LEVEL=HIGH to invoke vectorization. See chapter 13 for more information on vectorization and how to interpret the vectorizer messages.

RUNTIME_CHECKS (RC)

The RUNTIME_CHECKS parameter selects runtime range checking of subscripts and substrings. This parameter allows you to select multiple options. Options are:

NONE

Causes no options to be selected.

R

Selects runtime range checking for character substring expressions. If a character substring expression would cause the substring to exceed the bounds declared by the CHARACTER statement, an informative diagnostic is issued and execution continues.

S

Selects runtime range checking for subscript expressions and array conformability. If a subscript expression would cause the subscript to exceed its declared dimension bounds, an informative diagnostic is issued and execution continues. If arrays in assignment and WHERE statements are not conformable, an informative diagnostic is issued and execution continues.

ALL

Selects both the R and S options.

If RUNTIME_CHECKS is omitted, NONE is used.

NOTE

If RUNTIME_CHECKS specifies R, S, or ALL, and OPTIMIZATION_LEVEL=HIGH and VECTORIZATION_LEVEL=HIGH are also selected, the RUNTIME_CHECKS parameter is ignored.

STANDARDS_DIAGNOSTICS (SD)

The STANDARDS_DIAGNOSTICS parameter specifies whether the use of non-ANSI source statements is to be diagnosed and if so, how severely. Options are:

NONE

Nonstandard usages are not diagnosed.

INFORMATIONAL (I)

Nonstandard usages are treated as informational errors.

WARNING (W)

Nonstandard usages are treated as warning errors.

FATAL (F)

Nonstandard usages are treated as fatal errors.

If STANDARDS_DIAGNOSTICS is omitted, NONE is used.

See the ERROR_LEVEL parameter for descriptions of informational, warning, and fatal errors.

STATUS

The STATUS parameter specifies an SCL status variable to be set by the compiler with information about errors that occurred during compilation. You must create the status variable before specifying it. For example:

```
/var
var/s_variable_name: status
var/varend
```

These statements create a status variable named s_variable_name.

The severity level of errors for which information is to be returned is determined by the TERMINATION_ERROR_LEVEL parameter.

For more information about the status variable, see the STATUS parameter in the NOS/VE System Usage manual.

If STATUS is omitted, no error status information is returned.

The STATUS parameter must be specified by name.

TARGET_MAINFRAME (TM)

The TARGET_MAINFRAME parameter specifies the kind of mainframe that the object code is generated for. This parameter is only significant when the OPTIMIZATION_LEVEL parameter specifies HIGH. Options are:

C180_VECTOR (C180V)

The object code is optimized, if possible, to take advantage of the vector-processing capabilities present on CYBER 990 and CYBER 995 Class mainframes.

Object code compiled with this option and OPTIMIZATION_LEVEL=HIGH can be executed only on CYBER 990 or 995 Class mainframes.

Object code compiled with this option and OPTIMIZATION_LEVEL=LOW or DEBUG can be executed on any mainframe.

C180_MODEL_INDEPENDENT (C180MI)

The object code does not take advantage of mainframe specific hardware. The object code can be run on any mainframe, regardless of the optimization level.

C180_CURRENT_MAINFRAME (C180CM)

The object code is generated for use on the machine on which compilation occurs. If the compiler is on a CYBER 990 or 995 Class mainframe, then TARGET_MAINFRAME=C180V is selected. Otherwise, TARGET_MAINFRAME=C180MI is selected.

The selection (C180V or C180MI) used on your program is listed along with the other compilation parameters at the end of the source listing.

If TARGET_MAINFRAME is omitted, C180_CURRENT_MAINFRAME is used.

If you select OPTIMIZATION_LEVEL=LOW or DEBUG, the object code is generated for use on any mainframe, regardless of what the TARGET_MAINFRAME parameter specifies.

On CYBER 990 and 995 Class mainframes, the object code for TARGET_MAINFRAME=C180CM and TARGET_MAINFRAME=C180V is identical. Similarly, on any mainframe other than a CYBER 990 or 995 Class, the object code for TARGET_MAINFRAME=C180CM and TARGET_MAINFRAME=C180MI is identical.

For Better Performance

Be sure to use the TARGET_MAINFRAME=C180_VECTOR option for code that is going to be executed on a CYBER 990 or 995 Class mainframe.

TERMINATION_ERROR_LEVEL (TEL)

The TERMINATION_ERROR_LEVEL parameter specifies the minimum error severity level for which the compiler is to return abnormal status. The status code is returned in the SCL variable specified by the STATUS parameter after compilation completes. Information about errors having the specified severity or higher severity is returned. Refer to the ERROR_LEVEL parameter for an explanation of the error severity levels. Options are:

INFORMATIONAL (I)

Abnormal status is returned for informational, warning, fatal, and catastrophic errors.

WARNING (W)

Abnormal status is returned for warning, fatal, and catastrophic errors.

FATAL (F)

Abnormal status is returned for fatal and catastrophic errors.

CATASTROPHIC (C)

Abnormal status is returned for catastrophic errors only.

If TERMINATION_ERROR_LEVEL is omitted, FATAL is used.

VECTORIZATION_LEVEL (VL, VECTORIZATION, VEC)

The VECTORIZATION_LEVEL parameter specifies the vectorization level performed by the compiler. Options are:

NONE

No vectorization is performed.

HIGH

A high level of vectorization is performed, resulting in faster execution time but slower compilation time.

If VECTORIZATION_LEVEL is omitted, NONE is used.

You can use the REPORT_OPTIONS parameter to create a report containing implicit vectorization messages.

See chapter 13, Vectorization, for more information on how to interpret implicit vectorizer messages. This parameter is ignored if OPTIMIZATION_LEVEL=LOW or DEBUG or TARGET_MAINFRAME=C180MI is specified.

VECTOR_FORTRAN Command Examples

Following are some examples of VECTOR_FORTRAN commands.

Example:

```
/vector_fortran input=afile binary_object=bfile error_level=fatal
```

This statement selects the following options:

INPUT=AFILE
Source statements are read from file AFILE.

BINARY_OBJECT=BFILE
Object code is written to file BFILE.

ERROR_LEVEL=FATAL
Fatal and catastrophic diagnostics are written to the output listing file.

All other parameters assume default options. (See table 12-1.)

Example:

```
/vector_fortran myprog optimization_level=high
```

This statement selects the highest level of optimization of the program in file MYPROG. All other parameters assume default values. (See table 12-1.)

Example:

```
/vector_fortran
```

This statement selects default values for all parameters. (See table 12-1.)

Example:

```
/vector_fortran i=(afile, bfile, cfile) b=binf da=all
```

This statement selects the following options:

I=(AFILE, BFILE, CFILE)
Source statements are read from files AFILE, BFILE and CFILE.

B=BINF
The binary object code is written to file BINF. The binary object code is generated from all three input files into one file.

DA=ALL
Allows you to use the Debug utility as you execute your program.

Example:

```
/vecf i=myfile  ro=b  vl=high
```

This statement compiles the program in MYFILE using the implicit vectorization capabilities of the compiler. The RO parameter selects brief mode implicit vectorization messages to appear on the listing file. The messages indicate the level of vectorization that was achieved.

Example:

```
/vfortran
?       PROGRAM TEST
?       READ *, J
?       PRINT *, J/2
?       END
?*EOI
/
```

This example reads the source input from the terminal (assuming $INPUT is connected to the terminal) and then compiles it. To terminate terminal input, and begin program compilation, enter *EOI (in uppercase) after the prompt.

Example:

```
/vfortran i=$user.in1 vl=high ol=high tm=c180v
```

This example compiles the program in file $USER.IN1. The VECTORIZATION_LEVEL and OPTIMIZATION_LEVEL parameters select the highest level of optimization available on a class 990 or 995 mainframe.

Compiler Output Listing

The listing produced by the FORTRAN Version 2 compiler is controlled by the LIST_OPTIONS parameter on the VECTOR_FORTRAN command. You can select a complete listing or particular portions of the listing, or you can completely suppress the listing by specifying LIST_OPTIONS=NONE. If you supplied a list of file references for the INPUT parameter, the format of the compiler output listing is the same as if you had combined the files into a single file.

The following paragraphs describe the output listing options.

Source Listing

The source listing includes all source lines submitted for compilation as part of the source input file. Listed lines are preceded by a sequence number, unless you specify otherwise through the LINE_NUMBER attribute for the source input file.

You can use the C$ LIST directive to suppress the listing of selected source lines. If you select the LIST_OPTIONS=SA option, C$ LIST directives are disregarded and all source lines are listed. The C$ LIST directive is described in chapter 11, C$ Directives.

If errors are detected during compilation, a descriptive message is printed in the source listing immediately after the statement containing the error.

Compilation Statistics

The compilation statistics follow the source listing. These statistics always appear in the compiler listing and are not affected by the LIST_OPTIONS parameter unless LIST_OPTIONS=NONE is specified. The compilation statistics consist of the error summary and the total compilation time.

The error summary lists the total number of each level of error that occurred. The error levels are:

Nonstandard diagnostics
Machine-dependent diagnostics
Trivial diagnostics (Informational diagnostics)
Warning diagnostics
Fatal diagnostics
Catastrophic diagnostics

If the LIST and ERROR parameters both specify the same file (or if $LIST and $ERRORS are connected to the same file), each error message, along with the line containing the error, is written twice.

The level summary is followed by the message

```
TOTAL CP SECONDS IN VFTN COMPILATION=n
```

where n is the decimal number of central processor seconds required to compile the program.

Reference Map

The reference map is a dictionary of all symbolic names appearing in a program unit. The names are grouped by class and listed alphabetically within the groups. The reference map follows the source listing and the diagnostic summary (if present). The map is divided into the following sections:

Symbolic constants map
Namelist map
Variables map
Common and equivalence map
Statement labels map
DO loops map
Entry points map
Procedures map
I/O units map
Unclassified names map

The content of the reference map is controlled by the LIST_OPTIONS parameter on the VECTOR_FORTRAN command. The LIST_OPTIONS selections that affect the reference map are:

LIST_OPTIONS=A

Lists all symbolic names in the source program and their attributes (such as size, data type, relative address, and so forth).

LIST_OPTIONS=R

Lists each symbolic name in the program unit and all references to the name.

LIST_OPTIONS=(A,R)

Combines the A and R options (lists the attributes and references for each symbolic name).

LIST_OPTIONS=M

Produces a DO loop map and common/equivalence map. This option automatically selects the A option; that is, LIST_OPTIONS=M is the same as LIST_OPTIONS=(M,A), and LIST_OPTIONS=(M,R) is the same as LIST_OPTIONS=(M,A,R).

General Format of Maps

Each class of symbolic name is preceded by a subtitle line that specifies the class and the properties listed. Formats for each symbol class are different, but most contain the following information:

- Properties of the symbolic name.
- References to the symbol (LIST_OPTIONS=R). All line numbers refer to the source line containing the statement in which the reference occurs. A line number can be suffixed by one of the following usage symbols, which describes how the symbolic name was referenced:

  /A

  Attribute: The reference defines a particular attribute, such as type or dimension.

  /D

  Declare: The reference declares the name as a dummy argument, an entry point name, an entity in common, a namelist group name, or label of a DO loop terminator.

  /I

  Input: Name appears as an iolist item whose value will be read but not written.

  /M

  Modify: Name appears in a statement that alters its contents (assignment statement, DO statement, and so forth).

  /P

  Parameter: Name appears as an actual argument.

  /R

  Read: Name is an iolist item in an input statement, a namelist group name in a READ statement, or an I/O unit which is read.

  /S

  Subscript: Name appears in a subscript expression.

  /W

  Write: Name is an iolist item in an output statement, a namelist group name in a WRITE, PRINT, or PUNCH statement, or an I/O unit which is written.

- Relative address within the program unit of the symbol. This address is given in the form

  section + offset

  where section is the name of the section containing the symbol and offset is the offset (decimal words) of the symbol within the section, relative to the first word of the section. In most cases, the section name will be a program, subprogram, or common block name, or one of the following symbols:

  $LITERAL

  Section of the compiled program containing constant data.

  $STACK

  Section containing variables that are allocated on the stack when the containing program unit is called.

  $PARAMETER

  A subset of the $STACK section containing parameter list variables allocated on the stack by the calling program.

  $STATIC

  Section containing variables that are statically allocated, are not in common, and are not in an explicitly named section.

  $REGISTER

  Variables not belonging to any memory section but existing only in a hardware register.

The following paragraphs describe the various sections of the reference map as they would appear for the full map, selected by LIST_OPTIONS=(M,A,R).

Variables Map

Variable names include local and COMMON variables and arrays and dummy arguments. The variables map has the following form:

```
--VARIABLES--
  -NAME----------------------------SECTION+OFFSET------------SIZE-PROPERTIES-------TYPE---------REFERENCES

   name                             sec+off                   size prop1/prop2      type          refs
     :
```

name

Variable name. Variables are listed in alphabetical order.

sec+off

Relative address of the variable.

If name is a dummy argument, it is listed as DUMMY ARGUMENT #n, where n is the position of the argument in the dummy argument list. If name is used only as a statement function dummy argument, it is listed as an STF DUMMY ARGUMENT.

size

For array names, this entry gives the total number of elements in the array. For nonarray names this entry is blank.

prop

Properties of the name, listed in the form prop1/prop2... . Each prop is one of the keywords:

UND

Variable has not been defined. A variable is defined if any of the following conditions hold:

Appears as an entity in common.

Is initialized in a DATA statement.

Appears on the left side of an assignment statement at the outermost parenthesis level.

Is the DO variable in a DO loop or I/O implied DO list.

Appears as a parameter in a subroutine or function call.

Appears in an input list.

Appears as the IOSTAT variable in an I/O statement.

Appears as a status variable in an INQUIRE statement.

Appears as an extended internal file in an ENCODE statement.

Appears as a standard internal file in a WRITE statement.

Appears as the destination name in a BUFFER IN statement.

Otherwise the variable is considered undefined. However, variables that are referenced before they are defined are not flagged.

EQV

Variable name is equivalenced.

SAV

Variable name has the SAVE property.

UNUSED

Name is not referenced in an executable statement, is not a statement function dummy argument, is not an entity in common, and does not appear as a DO variable in a DATA implied DO list.

S

Name appears only once in the entire program unit. (Check carefully for other names with similar spellings.)

type

Gives the type of the variable name. One of the values LOGICAL, INTEGER, BYTE, REAL, COMPLEX, DOUBLE, CHARACTER, or BOOLEAN.

For character names, the form is:

CHAR*n

For variables with specified length

CHAR*(*)

For variables with adjustable length

refs

References and definitions associated with the variable name; listed by line number. Appears only when R option is selected. May be suffixed by a usage symbol.

Symbolic Constants Map

The symbolic constants map lists information about constants assigned symbolic names by PARAMETER statements. This map has the following form:

```
SYMBOLIC CONSTANTS--
  -NAME-------------------------------TYPE-----------------------VALUE---REFERENCES

   name                               type                        value   refs
     :
```

name

Symbolic name as declared in the PARAMETER statement.

type

Data type of the name (same as type field in VARIABLES section).

value

Value assigned to the name in the PARAMETER statement.

refs

Source line number of statements referencing the constant; appears only if R option selected. Suffixed by a usage symbol (same as refs field in VARIABLES map).

Namelist Map

The namelist map lists information about namelist groups defined in the program unit. This map has the following form:

```
NAMELISTS--
  -NAME--------------------------------SECTION+OFFSET---------REFERENCES

   name                                   sec+off              refs
     :
```

name

Namelist group name.

sec+off

Relative address of the first word of the namelist group.

The symbol *NONE* indicates that the namelist group was not referenced in the program unit.

refs

Source line numbers of statements referencing the namelist group. (Appears only if R option selected.) Suffixed by a usage symbol:

/D

Namelist group is defined.

/R

Namelist is referenced in a READ statement.

/W

Namelist is referenced in a WRITE statement.

Common and Equivalence Map

The common and equivalence map lists information about common blocks and equivalence declarations within the program unit. This map is selected by the M option on the LIST_OPTIONS parameter. The common and equivalence map has the following form:

```
COMMON+EQUIVALENCE--

  /block/  SIZE=size units save

   item1 ... itemn
      :

--LOCAL EQUIVALENCES--

  item1 ... itemn
     :
```

Compiler Output Listing

name

Common block name.

size

Total number of words or characters occupied by the common block.

units

WORDS for a block containing only noncharacter items. CHARACTERS if the block contains character items.

save

SAVE if the block is saved; blank otherwise.

item

Describes the storage position of a variable or array. Each item has three fields:

name<first-last> or name<first:last>

name

Name of the variable or array.

first

Storage position within the block of the first element of name.

last

Storage position within the block of the last element of name.

First and last are given in decimal. The first position of a common block is numbered 1. If name is a noncharacter variable (occupies a single word), -last does not appear. If name is type character, first and last indicate character positions, and are separated by a colon. Otherwise, they indicate words and are separated by a hyphen.

Items that share storage positions because of equivalencing are enclosed in parentheses.

Statement Labels Map

The statement labels map lists information about all statement labels used in the program unit. This map has the following form:

```
--STATEMENT LABELS--
 -LABEL-SECTION+OFFSET--------PROPERTIES-------DEF--REFERENCES-

  label sec+off                prop           def  refs
   :
```

label

Statement label; labels are listed in numerical order.

sec+off

Relative address of the label. If an address cannot be assigned, one of the following symbols appears:

UNDEFINED

Statement label is not defined.

NO REFERENCES

Label is not referenced anywhere in the program unit. (This label can be removed from the source program.)

INACTIVE

Label has been deleted by optimization.

prop

One of the following:

FORMAT

Label appears in a FORMAT statement.

DO-TERM

Label appears in a DO statement.

NON-EX

Label appears on a nonexecutable statement. (If the label is referenced, a diagnostic is issued.)

blank

Label appears in the label field of an executable statement.

def

Source line number where label is defined. *UNDEF* if not defined.

refs

Source line numbers where the label is referenced. (Appears only when R option selected.) May be suffixed by a usage symbol.

DO Loops Map

The DO loops map lists information about all DO loops in the program unit, including implied DO lists in I/O statements. (Implied DO lists in DATA statements are not listed.) The loops are listed in order of their appearance in the program unit. The map appears only if you selected the M option. The DO loop map has the following form:

```
--DO LOOPS--
 -LABEL--SECTION+OFFSET---------PROPERTIES---------INDEX---------------------------FROM---TO

   label  sec+off                  prop             index                           first  last
     :
```

label

Label of final statement; I/O for implied DO lists in input/output statements.

sec+off

Relative address of the first statement of the loop.

prop

One of the following symbols, describing properties of the loop:

OPEN

Loop can be reentered from outside its range.

EXIT

Loop contains references to statement labels outside its range.

XREF

Loop contains references to an external subprogram, including compiler-generated references to all library subprograms except those used to do character moves or compares.

OUTER

Loop contains nested loops.

index

Name of the DO variable.

from

Line number of the first statement of the loop.

to

Line number of the last statement of the loop.

Entry Points Map

The entry points map lists information about subprogram names appearing on ENTRY statements. This map has the following form:

```
--ENTRY POINTS--
 -NAME-------------------------------SECTION+OFFSET-----------ARGS-REFERENCES-

  name                                  sec+off                 args refs
    :
```

name

Entry point name as defined in the source program.

sec + off

Relative address assigned to the entry point.

args

Number of dummy arguments in the ENTRY statement; blank if no arguments.

refs

Source line numbers of ENTRY or RETURN statements. (Produced only if the R option is selected.) May be suffixed by a usage symbol.

Procedures Map

The procedures map lists names of all functions and subroutines called from the program unit, names declared in an EXTERNAL statement, and names of intrinsic and statement functions appearing in the program unit. Implicit external references generated by certain FORTRAN statements, such as input/output statements, are not listed. The procedures map has the following form:

```
--PROCEDURES--
 -NAME------------------------------TYPE----------ARGS-CLASS-----REFERENCES

  name                              type          args class     refs
    :
```

name

Subroutine or function name.

type

Data type of function result; blank if class is SUBROUTINE or UNKNOWN; GENERIC for generic intrinsic functions.

args

Number of arguments; VAR if number is variable (intrinsic functions such as MAX and MIN). UNKNOWN if class is UNKNOWN.

class

One of the following:

DUMMY FUNC

Name is a dummy argument used as a function name

DUMMY SUBR

Name is a dummy argument used as a subroutine name

SUBROUTINE

Name is used as a subroutine name

FUNCTION

Name is used as an external function name

FUNC+SUBR

Name used as both a function name and subroutine name (gives a warning message)

INTRINSIC

Name is an intrinsic function name

STAT FUNC

Name is used as a statement function name

UNKNOWN

Class cannot be determined. (Appears if name is used in an EXTERNAL statement.)

refs

Line number where name is referenced. May be suffixed by a usage symbol.

Input/Output Units Map

The input/output units map lists all constant unit designators referenced in the program unit. This map has the following format:

```
--IO UNITS--
 -NAME-----ALIAS---PROPERTIES------------REFERENCES-

  name     alias    prop                   refs
    :
```

name

Value of the unit designator. If the value is an integer in the range 0 through 999, then name has the form TAPEn.

alias

The name of the alternate unit specified by an alternate unit specification on the PROGRAM statement or by default (in the case of an implied unit for which no alternate unit was specified on the PROGRAM statement). This field is blank if the unit does not have an alias.

prop

Type of I/O operation for which the unit is used:

FMT

Formatted operation

SEQ

Sequential operation

DIR

Direct access operation

BUF

Buffer I/O operation

AUX

Auxiliary I/O statement

Multiple symbols are separated by slashes.

refs

Source line number of statements referencing the unit; appears only if R option selected. May be suffixed by a usage symbol.

Unclassified Names Map

The unclassified names map lists all names appearing in the program unit that could not be classified. In many cases, these names are the result of programming errors such as misspellings, and should be checked carefully. The unclassified names map has the following form:

```
--UNCLASSIFIED NAMES--
 -NAME---------------------------------PROPERTIES-------REFERENCES

  name                                  prop             refs
    :
```

name

Name as it appeared in the source program.

prop

One of the following:

UNUSED

Name is not referenced in an executable statement, is not a statement function dummy argument, is not an entity in common, and does not appear as a DO variable in a DATA implied DO list.

S

Name appears only once in the entire program unit. (Check carefully for other names with similar spellings.)

refs

Source line number where the name is referenced. (Produced only if the R option is selected.) May be suffixed by a usage symbol.

Execution Command

An execution command loads and executes a compiled program. The execution command has the form:

file *p* ... *p*

file

Specifies the file containing the object program to be executed.

p

Optional parameter to be passed to the executing program.

You can also use the EXECUTE_TASK command to begin execution of a program. This command has the form

EXECUTE_TASK *file params*

file

Specifies the name of the object file to be executed.

params

List of parameters.

See the NOS/VE Object Code Management manual for a detailed description of the EXECUTE_TASK command. (See also the NOS/VE Object Code Management manual for information on the system loader.)

The name of the object file is established by the BINARY_OBJECT parameter on the VECTOR_FORTRAN command. If you omit this parameter, the file name defaults to $LOCAL.LGO.

The execution command causes the system loader to load the compiled program into memory, perform the required linking and address relocation, and initiate execution of the loaded program. If any parameters are specified on the execution command, they are passed to the program.

The optional parameters on the execution command provide a method of passing values to the program. The parameter list must conform to the format for parameter lists described in the NOS/VE System Usage manual. Three classes of parameters can appear on the execution call command: predefined parameters (STATUS and $PRINT_LIMIT), file names to be used for file name substitution, and user-defined SCL parameters. You can specify either file names for file name substitution or SCL parameters, but not both, on an execution command.

Parameters on an execution command can be specified by name (in the form parameter name=value) or positionally (parameter name= omitted). When you specify a parameter positionally, you must indicate the position of any omitted parameters that precede the specified parameter by commas; the first omitted parameter is indicated by two successive commas, and each additional omitted parameter is indicated by an additional comma. Thus, if *n* parameters are omitted, $n+1$ commas are required.

For Better Performance

You can use the AFTERBURN_OBJECT_TEXT command to expand referenced subprograms inline after they are compiled, thereby reducing the call/return overhead of your program. The AFTERBURN_OBJECT_ TEXT command is described in the NOS/VE Object Code Management manual.

$PRINT_LIMIT Parameter

The $PRINT_LIMIT parameter appears on the execution command as follows:

file $PRINT_LIMIT=lim or **file $PL=lim**

where file specifies the file containing the compiled object code and lim is the desired print limit. This parameter specifies the maximum decimal number of print lines that the executing program can write to files $OUTPUT and $ERRORS. If the $PRINT_LIMIT parameter is specified positionally, it is the next to last parameter on the execution command.

Example:

```
/lgo $print_limit=10000
```

This command sets the runtime print limit to 10000 lines.

NOTE

This parameter is only valid for batch jobs; it is not valid in interactive sessions.

STATUS Parameter

The STATUS parameter specifies a System Command Language variable to be used for the error status code returned by the system when runtime errors occur. The STATUS parameter appears on the execution command as follows:

file STATUS=var

where file specifies the object file name and var is a variable to receive the error status code. The STATUS variable consists of the following fields:

Normal field

Returns the logical value false if errors occurred, or true if no errors occurred. Information is returned in the identifier, condition, and text fields only if the normal field contains the value false.

Condition field

Returns the error number.

Text field

A field of length 256 characters in which SCL places a string containing delimited substrings to be substituted into the message template for the particular error.

The STATUS parameter is positionally the last parameter on the execution command. This parameter must be specified by name.

Example:

```
/var
 var/ieer:status
 var/varend
/lgo status=ierr
```

These statements define variable IERR to be the error status variable.

NOTE

You can use the ABORT_FILE parameter on the EXECUTE_TASK_COMMAND to specify a file containing Debug commands. The commands are used if the program is not executed in Debug mode. This file can be positioned. The ABORT_FILE parameter has the form:

ABORT_FILE=file reference

Example:

```
/execute_task file=my_program abort_file=$user.debug_commands
```

User-Defined System Command Language Parameters

You can specify parameters on the execution command that are accessible within the executing program. These parameters provide a method of passing information between an executing program and NOS/VE. The parameter names and values are accessed by using the Execution Command Parameter Subprograms described in chapter 10, NOS/VE and Utility Subprograms. If any user-defined SCL parameters are to appear on an execution command, those parameters must be defined by the C$ PARAM directive within the program to be executed.

NOTE

A given execution command cannot contain both SCL parameters and parameters for file name substitution.

File Name Substitution

FORTRAN provides a method of substituting file names at execution time. File names specified on the execution command are substituted for file names associated with unit names declared on the PROGRAM statement.

Unit names declared on the PROGRAM statement are associated with a default file of the same name unless you substitute a different name. For units INPUT and OUTPUT, the default files are $INPUT and $OUTPUT, respectively. For other units, the default files have the same name as the unit. For example, with the PROGRAM statement

```
PROGRAM TEST(INPUT, OUTPUT, TAPE1, TAPE2)
```

The default runtime file names are:

$INPUT

$OUTPUT

TAPE1

TAPE2

Specifying unit names on the PROGRAM statement is optional; the same unit names would exist if the statement PROGRAM TEST were used. However, default file names associated with the unit names on the PROGRAM statement can be changed for a particular run by using the method of file name substitution.

Parameters to be used for file name substitution have the same format as SCL parameters; however, a C$ PARAM directive in the program is not permitted. Each unit declaration on the PROGRAM statement defines a valid parameter that can appear on the execution command. If you specify a parameter and value on the execution command in the form

parameter=value

the file name indicated by value is substituted for the file name associated with the unit name indicated by parameter. If only value is specified, the file name specified by value is substituted for the file associated with the unit name having the corresponding position on the PROGRAM statement. For example, if a program begins with the statement

```
PROGRAM TEST(INPUT, OUTPUT, TAPE1, TAPE2)
```

then the following execution commands have the same effect:

```
/lgo,,,myfile urfile

/lgo tape1=myfile tape2=urfile
```

Each command causes the following default associations:

Unit Name	File Name Used
INPUT	$INPUT
OUTPUT	$OUTPUT
TAPE1	MYFILE
TAPE2	URFILE

The commas on the first LGO command are required to indicate the omitted INPUT and OUTPUT parameters.

If an alternate unit name is specified in the PROGRAM statement, that name can be used in place of the parameter name on the execution command. For example, if a program uses the statement

```
PROGRAM TEST (INPUT, TAPE5=INPUT, OUTPUT)
```

then the following execution commands are equivalent:

```
/lgo myfile urfile,,status=check

/lgo tape5=myfile output=urfile status=check
```

The file substitutions are:

Unit Name	File Name Used
INPUT	MYFILE
TAPE5	MYFILE
OUTPUT	URFILE

In the first command, the status variable CHECK is specified positionally (it is the last parameter on the command); the omitted $PRINT_LIMIT parameter is indicated by two successive commas. In the second command, the status variable is a specified parameter name.

The default unit/file associations can be overridden by file name specifications in OPEN statements.

Vectorization 13

What is Vectorization? 13-2

Why is Vectorization Faster? 13-4

What is Explicit Vectorization? 13-6

What is Implicit Vectorization? 13-9

How the Compiler Selects Statements for Implicit Vectorization 13-11

Statements That Cannot be Vectorized Implicitly 13-13

Vector Operations 13-15
- Scattering and Gathering 13-16
- The Sequence Vector 13-18
- Sum Reduction 13-19
- Scalar Expansion 13-20
- Broadcasting 13-21
- Intrinsic Function Promotion 13-22
- Array Sectioning 13-23
- Loop Interchange 13-24
- Induction Variable Elimination 13-25
- Stripmining 13-26
- Loop Collapse 13-28

Controlling Implicit Vectorization with C$ Directives 13-29
- C$ ASSUME(NORECUR) 13-29
- C$ ASSUME(MAXITER=c) 13-29
- C$ CONTROL(p) 13-30

Producing the Implicit Vectorization Report 13-31

Implicit Vectorization Messages 13-35
- Message Number: 4101 13-35
- Message Number: 4102 13-35
- Message Number: 4103 13-35
- Message Number: 4104 13-35
- Message Number: 4120 13-35
- Message Number: 4122 13-36
- Message Number: 4124 13-36
- Message Number: 4126 13-36
- Message Number: 4200 13-36
- Message Number: 4201 13-37
- Message Number: 4210 13-37
- Message Number: 4220 13-37
- Message Number: 4222 13-37
- Message Number: 4224 13-37
- Message Number: 4226 13-37
- Message Number: 4228 13-38
- Message Number: 4230 13-38
- Message Number: 4240 13-38
- Message Number: 4310 13-38
- Message Number: 4312 13-39
- Message Number: 4320 13-39
- Message Number: 4322 13-39

Message Number: 4324 13-39
Message Number: 4326 13-40
Message Number: 4328 13-40

Vectorization 13

This chapter presents an introduction to vectorization using NOS/VE FORTRAN Version 2 on the 990 and 995 class mainframes. Vectorization allows a program to execute faster because it performs operations on vectors (sets of values) instead of scalars (single values). There are two types of vectorization discussed in this chapter, explicit and implicit.

The FORTRAN Version 2 compiler can automatically vectorize a program using the vector-processing hardware and software of the CYBER 180 Model 990/995. This is implicit vectorization. You can also use explicit vector notation in your program to take advantage of the vector-processing hardware and software on the Model 990/995. Explicit vector notation includes array sections, array expressions, array assignment statements, array-valued intrinsic functions, and WHERE statements.

This chapter discusses the following:

- What is vectorization?
- Why is vectorization faster?
- What is explicit vectorization?
- What is implicit vectorization?
- Statements that cannot be vectorized implicitly
- Types of implicit vectorization
  - Scattering and gathering
  - The sequence vector
  - Sum reduction
  - Scalar expansion
  - Broadcasting
  - Intrinsic function promotion
  - Array sectioning
  - Loop interchange
  - Induction variable elimination
  - Stripmining
  - Loop collapse
- Producing the implicit vectorization report
- Controlling implicit vectorization with the C$ directives
- The implicit vectorization messages, ordered by number

What is Vectorization?

Vectorization is the process of transforming scalar operations into vector operations. Vector operations are performed with vectors. To a mathematician, a vector is a set of N numbers uniquely defining a distance and a direction in an N-dimensional space. A FORTRAN programmer generally uses the term vector of length N to mean any one-dimensional set of N numbers.

Arrays and array sections can be thought of as vectors. For example, the following statements define a 5-element vector and store some values into it:

```
DIMENSION A(5)
DATA A /6.0, 1.0, 2.0, 9.0, 7.0/
```

The vector can be pictured as follows:

A(1)	A(2)	A(3)	A(4)	A(5)
6.0	1.0	2.0	9.0	7.0

A vector can be contrasted with a scalar. A scalar is simply a single value, such as a constant, variable, or array element. The familiar arithmetic operations in your existing FORTRAN programs are known as scalar operations. In the following example, the variables R and S are scalars. The addition of R and S is a scalar operation, and the result is stored in the scalar variable T:

```
DATA R, S/1.0, 2.0/
      ⋮
T=R+S
```

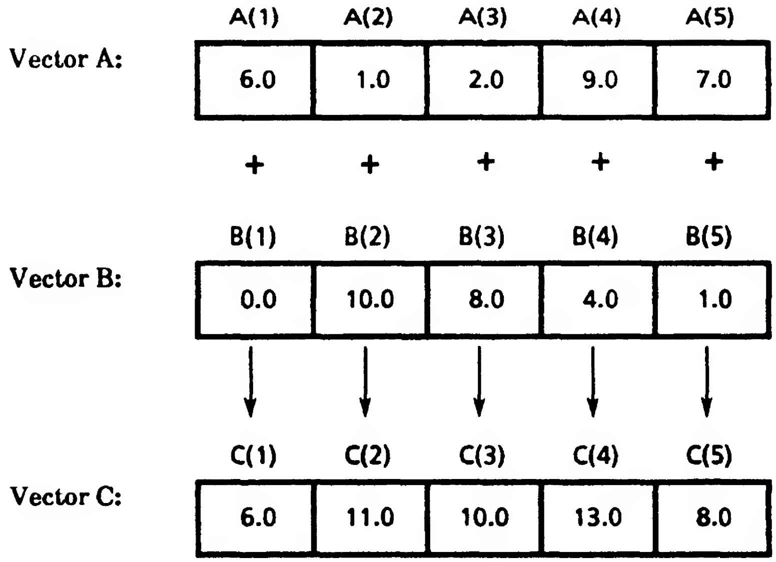
Scalar operations involve single-valued operands and calculate single-valued results.

Vector operations, however, operate on vectors and calculate results for vectors. Vectors consist of more than one element. The following example shows an array operation that can be performed internally as a single vector operation:

```
    DIMENSION A(10), B(10), C(10)
    DATA A/6.0, 1.0, 2.0, 9.0, 7.0/
    DATA B/0.0, 10.0, 8.0, 4.0, 1.0/
          ⋮
    DO 10 I=1, 10
       C(I)=A(I)+B(I)
10  CONTINUE
```

This program defines three arrays, A, B, and C. The contents of arrays A and B are added, and the result is stored in array C. The addition can be performed as vector addition because A and B can be represented as vectors in the machine. In a vector addition, elements of one vector are added to the corresponding elements of another vector.

The vector addition can be pictured as follows:



Thus, the result of this example is a vector having the values 6.0, 11.0, 10.0, 13.0, and 8.0.

Why is Vectorization Faster?

Vector operations on the CYBER 180 Model 990 are performed using a pipeline type of processor. To see why the pipeline processor executes faster, consider the following DO loop, which adds the number 4.0 to each element of a 10-element array A and stores the results in array B:

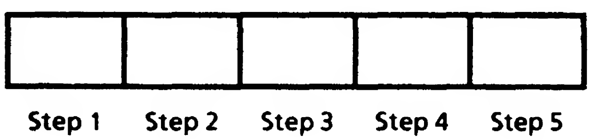
```
      DIMENSION A(10), B(10)
            :
      DO 2 I=1,10
        B(I)=A(I)+4.0
   10 CONTINUE
```

Consider a single calculation within this DO loop, the calculation B(1)=A(1)+4.0. The FORTRAN Version 2 compiler translates this statement into a sequence of machine instructions. To simplify the example, we assume a hypothetical computer on which the assignment statement B(1)=A(1)+4.0 is translated into the following machine instructions:

```
LOAD      A(1)
ADD       4.0
STORE     B(1)
LOOP
```

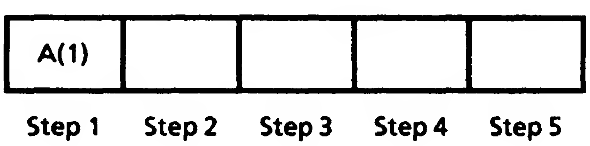
Now focus on the ADD instruction. On a scalar processor, the ADD instruction for a particular operand must be complete before the ADD for the next operand can begin. For example, the ADD for A(1) must be complete before the ADD for A(2) can begin.

On a pipeline processor, however, the ADD is divided into a sequence of steps. For our hypothetical machine, we assume five steps, with each step requiring one cycle of execution time. Thus, the ADD instruction can be pictured as follows:



Each box represents one step of the ADD instruction.

In scalar processing, all steps must be complete for an operand before processing of the next operand can begin; a single ADD takes 5 cycles. With pipeline processing, after a step is complete, that step can process the next operand. In our example, at the beginning of the first cycle, the first step of the ADD instruction begins processing A(1):



When step 1 is complete for A(1), A(1) moves on to step 2, and step 1 begins for A(2):

A(2)	A(1)			
Step 1	Step 2	Step 3	Step 4	Step 5

At the end of cycle 2, A(1) moves on to step 3, A(2) moves to step 2, and step 1 begins processing the next operand. Following are diagrams for cycles 3, 4, and 5:

Cycle 3:

A(3)	A(2)	A(1)		
Step 1	Step 2	Step 3	Step 4	Step 5

Cycle 4:

A(4)	A(3)	A(2)	A(1)	
Step 1	Step 2	Step 3	Step 4	Step 5

Cycle 5:

A(5)	A(4)	A(3)	A(2)	A(1)
Step 1	Step 2	Step 3	Step 4	Step 5

At cycle 5, five operands are being processed by the ADD instruction. At cycle 6, the operation A(1)+4.0 is complete and the result is ready to be stored into B(1):

A(6)	A(5)	A(4)	A(3)	A(2)	
Step 1	Step 2	Step 3	Step 4	Step 5	→ A(1) + 4.0 → B(1)

Subsequently, an ADD is completed every cycle. The hardware that allows for processing multiple operands by a single instruction is known as a pipeline.

What is Explicit Vectorization?

Explicit vectorization is vectorization that you control by putting explicit vectorization statements in your source program. If you are compiling and executing your program on a class 990 or 995 mainframe, the compiler tries to generate vector instructions from the explicit vectorization statements. Explicit vectorization statements can be used on a non-990 or 995 mainframe, however, no vector processing occurs. See the TARGET_MAINFRAME parameter in chapter 12, Compiling and Executing, for more information.

The explicit vectorization statements and constructs provided by FORTRAN Version 2 are:

- Array-valued assignment statements
- Array sections
- WHERE statements and logical WHERE structures
- Array-valued intrinsic functions

Array-valued assignment statements allow you to assign values to an entire array without the use of a DO loop. For example,

```
DIMENSION ARRAY(200)
ARRAY = 0.0
```

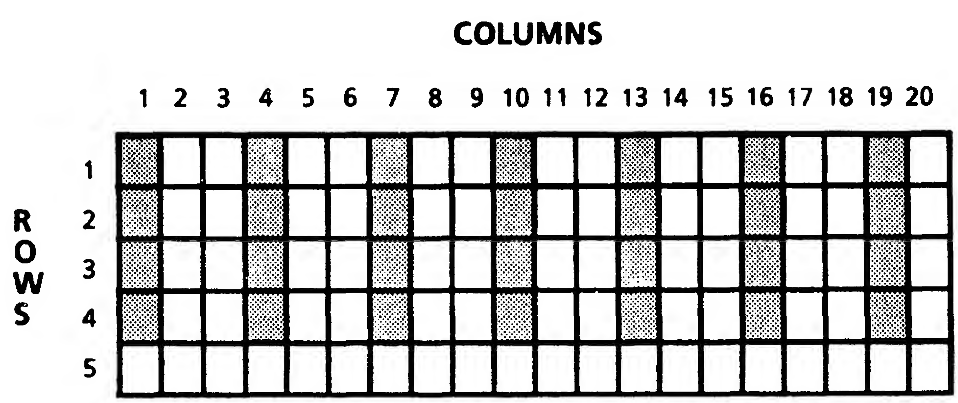
The array assignment statement assigns the value 0.0 to every element in ARRAY.

The use of array sections lets you control which parts of an array are being referenced. For example:

```
DIMENSION ARRAY_A(5,20)
ARRAY_A(1,1:20:3) = 1.5
```

The assignment statement assigns a value to every third element of the second dimension of the array.

The following diagram indicates, by shading, the elements in the array section ARRAY_A(1:4,1:20:3):



For more information about array-valued assignment and array sections, see chapter 3, Arrays, and chapter 4, Expressions and Assignment Statements.

The WHERE statement controls array assignment by using a logical, or masking array.

Example:

```
LOGICAL ARRAY_LOGIC(3,4)
DIMENSION  ARRAY_B(3,4), ARRAY_C(3,4)
WHERE (ARRAY_LOGIC) ARRAY_B=ARRAY_C
```

The WHERE statement assigns corresponding elements of ARRAY_C to ARRAY_B for every corresponding element of the logical array ARRAY_LOGIC with a true (.TRUE.) value. The logical array is used as a mask, controlling the assignment of values. Elements in ARRAY_B that correspond to elements in the logical array with a value of .FALSE. are unchanged.

Example:

Before the WHERE statement is executed:

ARRAY_B

4	4	4	4
4	4	4	4
4	4	4	4

ARRAY_LOGIC

T	F	T	T
F	T	F	T
F	T	T	T

ARRAY_C

7	7	7	7
7	7	7	7
7	7	7	7

After the WHERE statement is executed:

ARRAY_B

7	4	7	7
4	7	4	7
4	7	7	7

ARRAY_LOGIC

T	F	T	T
F	T	F	T
F	T	T	T

ARRAY_C

7	7	7	7
7	7	7	7
7	7	7	7

You can also use a WHERE block, which allows you to control assignment using both .TRUE. and .FALSE. values in the masking array. For example:

```
WHERE (ARRAY_LOGIC)
       ARRAY_B=ARRAY_C
ELSEWHERE
       ARRAY_D=0.0
ENDWHERE
```

The assignment of values to ARRAY_B is controlled by .TRUE. values in ARRAY_LOGIC while the assignment of values to ARRAY_D is controlled by .FALSE. values in ARRAY_LOGIC.

Masking arrays are also used in the intrinsic functions SUM, MERGE, and PRODUCT.

All of the elemental intrinsic functions accept array-valued arguments. For example, you could assign the square root of every element of one array to every element of another array (as long as the arrays are compatible) as follows:

```
ARRAY_A=SQRT(ARRAY_B)
```

The array-processing intrinsic functions can manipulate matrices. Some of the functions also allow a mask argument, which controls the processing of other array arguments.

The WHERE statement controls array assignment by using a logical, or masking array.

Example:

```
LOGICAL ARRAY_LOGIC(3,4)
DIMENSION  ARRAY_B(3,4), ARRAY_C(3,4)
WHERE (ARRAY_LOGIC) ARRAY_B=ARRAY_C
```

The WHERE statement assigns corresponding elements of ARRAY_C to ARRAY_B for every corresponding element of the logical array ARRAY_LOGIC with a true (.TRUE.) value. The logical array is used as a mask, controlling the assignment of values. Elements in ARRAY_B that correspond to elements in the logical array with a value of .FALSE. are unchanged.

Example:

Before the WHERE statement is executed:

ARRAY_B	ARRAY_LOGIC	ARRAY_C
4 4 4 4	T F T T	7 7 7 7
4 4 4 4	F T F T	7 7 7 7
4 4 4 4	F T T T	7 7 7 7

After the WHERE statement is executed:

ARRAY_B	ARRAY_LOGIC	ARRAY_C
7 4 7 7	T F T T	7 7 7 7
4 7 4 7	F T F T	7 7 7 7
4 7 7 7	F T T T	7 7 7 7

You can also use a WHERE block, which allows you to control assignment using both .TRUE. and .FALSE. values in the masking array. For example:

```
WHERE (ARRAY_LOGIC)
      ARRAY_B=ARRAY_C
ELSEWHERE
      ARRAY_D=0.0
ENDWHERE
```

The assignment of values to ARRAY_B is controlled by .TRUE. values in ARRAY_LOGIC while the assignment of values to ARRAY_D is controlled by .FALSE. values in ARRAY_LOGIC.

Masking arrays are also used in the intrinsic functions SUM, MERGE, and PRODUCT.

All of the elemental intrinsic functions accept array-valued arguments. For example, you could assign the square root of every element of one array to every element of another array (as long as the arrays are compatible) as follows:

```
ARRAY_A=SQRT(ARRAY_B)
```

The array-processing intrinsic functions can manipulate matrices. Some of the functions also allow a mask argument, which controls the processing of other array arguments.

What is Implicit Vectorization?

Implicit vectorization is vectorization that the compiler performs on DO loops in your program when you select VECTORIZATION_LEVEL= HIGH at compile time. You don't need to make any changes to your program for implicit vectorization to work. Implicit vectorization is performed by a component of the compiler called the vectorizer.

FORTRAN programs frequently contain many DO loops that perform operations on arrays. If the arrays are large, the loops can be quite time-consuming. The vector processing capability of the CYBER 180 Model 990 provides a way of making array operations within DO loops considerably faster.

For example, if a program contained the statements

```
      DIMENSION ARRAY(100)
      DO 20 I=1, 100
        ARRAY(I)=0.0
   20 CONTINUE
```

the compiler would vectorize the statements internally, that is, it would treat the DO loop as the array assignment statement

```
      ARRAY=0.0
```

To see the results of implicit vectorization on your program, you can generate an implicit vectorization report. The messages in the report indicate which statements were vectorized. To generate the report, compile your program with the REPORT_OPTIONS parameter specifying BRIEF or FULL, depending on the level of messages you want to see.

The following implicit vector operations are described later in this chapter:

Scattering and gathering

Using the sequence vector

Sum reduction

Scalar expansion

Broadcasting

Intrinsic function promotion

Array sectioning

Loop interchange

Induction variable elimination

Loop collapse

Stripmining

Implicit vectorization can cause the reordering of compiler-generated instructions. The reordering is apparent in object listings.

Because of the startup overhead involved in vector processing, vector operations may not be faster than scalar operations for short vectors or loops that iterate few times. Loops that have an iteration count of five or less, known at compile time, are not eligible for vectorization. If the iteration count is not known at compile time, the iteration count is assumed to be greater than five.

However, vector operations are almost always faster for longer vectors; the longer the vectors, the greater the execution time savings. Also, the time savings achieved depends on the operations being performed.

How the Compiler Selects Statements for Implicit Vectorization

To select automatic vectorization of your program, specify VECTORIZATION_LEVEL=HIGH (or VL=HIGH) on the VECTOR_FORTRAN compilation command. When you select VL=HIGH, you must also select OPTIMIZATION_LEVEL=HIGH (or OL=HIGH). Vectorization is not done when OPTIMIZATION_LEVEL=DEBUG or LOW. Specifying VECTORIZATION_LEVEL=HIGH usually results in faster execution time of the resulting object code, but it does increase compilation time.

During compilation, the vectorizer first scans the source code for DO loops. If there are no DO loops in your program, no vectorization occurs. If there are DO loops in your program, they must meet certain basic criteria before they are considered for vectorization. If the DO loop satisfies these three criteria, the vectorizer analyzes it further to determine if it can be vectorized:

1. DO variable must be of type integer (eight bytes)

2. If the iteration count can be determined at compilation time, it must be greater than five.

DO label [,] var = exp1, exp2 [,exp3]

•
•
•
•

3. No branches out of the loop (such as a GOTO or arithmetic IF statement) from within the body of the DO loop.

label final statement

Once the DO loop meets these criteria, it is a candidate for vectorization. However, some types of statements within the loop can still prevent vectorization of the DO loop. DO loops can be partially vectorized, that is, some statements are vectorized and others are not. For a description of statements that cannot be vectorized, see Statements that Cannot be Vectorized Implicitly in the next section.

Although a DO loop containing a branch out of the loop or a potential branch out of the loop cannot be vectorized, surrounding loops can be vectorized.

Here is an example of a DO loop that is ineligible for vectorization:

```
      DO 20 I=1,200              !  This DO loop
        IF (I .GE. IVAR) GO TO 10 !  cannot
   20 A(I)=0.0                   !  be vectorized.
   10 PRINT 100,A
```

The GO TO in the logical IF could cause a side branch from within the body of the loop; therefore, this loop cannot be vectorized.

The data type of values used within DO loops affects the amount of vectorization that occurs. A statement containing a character operand cannot be vectorized. Full-word real, integer, and logical data are more likely to be vectorized. Double precision, boolean, and complex data can also be vectorized, but not as often as the above mentioned types.

Internal vector operations can be type-specific; that is, they work only for a certain data type. The descriptions of the internal vector operations in the Vector Operations section indicate any type-specific restrictions. If a statement contains a data type that cannot be fully vectorized, the vectorization report indicates that the DO loop containing the statement was partially vectorized.

Implicit conversion does not change the data type of the array. For example:

```
DIMENSION B(100)
B=1
```

The compiler assigns the real value 1.0 to each element of B because B is implicitly typed as a real variable.

To see the results of vectorization of your program, use the REPORT_OPTIONS parameter on the VECTOR_FORTRAN command as described in Producing the Implicit Vectorization Report in this chapter.

For Better Performance

Multidimensional arrays in DO loops should be used with care. Whenever possible, the innermost loop should iterate over the first subscript, the next innermost loop should iterate over the second subscript, and so forth. A loop that iterates in this way executes faster than one which iterates in some other order. This is because each reference to the array is made to the next closest array element (arrays are stored in column order). For example, the following statements do not reference elements in the order they are stored:

```
      DIMENSION A(20, 30, 40), B(20, 30, 40)
          :
      DO 10 I=1, 20
      DO 10 J=1, 30
      DO 10 K=1, 40
        A(I, J, K)=B(I, J, K)
   10 CONTINUE
```

The following example ensures that elements are referenced in the order they are stored and therefore executes faster:

```
      DIMENSION A(20,30,40), B(20,30,40)
          :
      DO 10 K=1,40
      DO 10 J=1,30
      DO 10 I=1,20
        A(I, J, K)=B(I, J, K)
   10 CONTINUE
```

The benefits of accessing arrays in storage order are achieved regardless of the VECTORIZATION_LEVEL specification.

If you specify ONE_TRIP_DO=ON and VECTORIZATION_LEVEL=HIGH, be sure that all DO loops in your program have a minimum trip count of one. If a DO loop that is being vectorized has a zero or negative minimum trip count, the vectorized version of the DO loop does not execute once as it does in the scalar version.

How the Compiler Selects Statements for Implicit Vectorization

To select automatic vectorization of your program, specify VECTORIZATION_LEVEL=HIGH (or VL=HIGH) on the VECTOR_FORTRAN compilation command. When you select VL=HIGH, you must also select OPTIMIZATION_LEVEL=HIGH (or OL=HIGH). Vectorization is not done when OPTIMIZATION_LEVEL=DEBUG or LOW. Specifying VECTORIZATION_LEVEL=HIGH usually results in faster execution time of the resulting object code, but it does increase compilation time.

During compilation, the vectorizer first scans the source code for DO loops. If there are no DO loops in your program, no vectorization occurs. If there are DO loops in your program, they must meet certain basic criteria before they are considered for vectorization. If the DO loop satisfies these three criteria, the vectorizer analyzes it further to determine if it can be vectorized:

1. **DO variable must be of type integer (eight bytes)**

2. **If the iteration count can be determined at compilation time, it must be greater than five.**

DO label [,] var = exp1, exp2 [,exp3]

•
•
•
•

3. **No branches out of the loop (such as a GOTO or arithmetic IF statement) from within the body of the DO loop.**

label final statement

Once the DO loop meets these criteria, it is a candidate for vectorization. However, some types of statements within the loop can still prevent vectorization of the DO loop. DO loops can be partially vectorized, that is, some statements are vectorized and others are not. For a description of statements that cannot be vectorized, see Statements that Cannot be Vectorized Implicitly in the next section.

Although a DO loop containing a branch out of the loop or a potential branch out of the loop cannot be vectorized, surrounding loops can be vectorized.

Here is an example of a DO loop that is ineligible for vectorization:

```
      DO 20 I=1,200              ! This DO loop
        IF (I .GE. IVAR) GO TO 10 ! cannot
   20 A(I)=0.0                   ! be vectorized.
   10 PRINT 100,A
```

The GO TO in the logical IF could cause a side branch from within the body of the loop; therefore, this loop cannot be vectorized.

The data type of values used within DO loops affects the amount of vectorization that occurs. A statement containing a character operand cannot be vectorized. Full-word real, integer, and logical data are more likely to be vectorized. Double precision, boolean, and complex data can also be vectorized, but not as often as the above mentioned types.

Internal vector operations can be type-specific; that is, they work only for a certain data type. The descriptions of the internal vector operations in the Vector Operations section indicate any type-specific restrictions. If a statement contains a data type that cannot be fully vectorized, the vectorization report indicates that the DO loop containing the statement was partially vectorized.

Implicit conversion does not change the data type of the array. For example:

```
DIMENSION B(100)
B=1
```

The compiler assigns the real value 1.0 to each element of B because B is implicitly typed as a real variable.

To see the results of vectorization of your program, use the REPORT_OPTIONS parameter on the VECTOR_FORTRAN command as described in Producing the Implicit Vectorization Report in this chapter.

For Better Performance

Multidimensional arrays in DO loops should be used with care. Whenever possible, the innermost loop should iterate over the first subscript, the next innermost loop should iterate over the second subscript, and so forth. A loop that iterates in this way executes faster than one which iterates in some other order. This is because each reference to the array is made to the next closest array element (arrays are stored in column order). For example, the following statements do not reference elements in the order they are stored:

```
      DIMENSION A(20, 30, 40), B(20, 30, 40)
            :
      DO 10 I=1, 20
      DO 10 J=1, 30
      DO 10 K=1, 40
        A(I, J, K)=B(I, J, K)
   10 CONTINUE
```

The following example ensures that elements are referenced in the order they are stored and therefore executes faster:

```
      DIMENSION A(20,30,40), B(20,30,40)
            :
      DO 10 K=1,40
      DO 10 J=1,30
      DO 10 I=1,20
        A(I, J, K)=B(I, J, K)
   10 CONTINUE
```

The benefits of accessing arrays in storage order are achieved regardless of the VECTORIZATION_LEVEL specification.

If you specify ONE_TRIP_DO=ON and VECTORIZATION_LEVEL=HIGH, be sure that all DO loops in your program have a minimum trip count of one. If a DO loop that is being vectorized has a zero or negative minimum trip count, the vectorized version of the DO loop does not execute once as it does in the scalar version.

Statements That Cannot be Vectorized Implicitly

The following statement types cannot be vectorized:

- A statement that causes an external reference; for example, a CALL statement, non-intrinsic function references, and all input/output statements cause external references.
- A statement that references elements shorter than eight bytes, such as 2- or 4-byte integers.
- A statement that references elements of type character.
- A statement or group of statements that is executed an indeterminate number of times; for example, statements controlled by a logical or arithmetic IF.
- A statement in a recurrence cycle. Recurrence cycles are described below.

A recurrence cycle exists when two or more statements depend on one another. This can occur when more than one iteration of a DO loop causes the same location in memory to be accessed. Recurrence cycles prevent vectorization because all statements in a recurrence cycle must be executed for a particular iteration of a DO loop before the next iteration of the loop can take place. In other words, these statements cannot be executed in parallel and therefore cannot be vectorized. For example:

```
      DO 10  I=1,N
         A(I)=B(I)+C(I)          Statement 1
         B(I+1)=A(I)             Statement 2
   10 CONTINUE
```

By unraveling the first three passes through the DO loop, we can more clearly see the recurrence cycle:

I=1 (1st iteration):

```
A(1) = B(1) + C(1)    Statement 1

B(2) = A(1)           Statement 2
```

I=2 (2nd iteration):

```
A(2) = B(2) + C(2)    Statement 1

B(3) = A(2)           Statement 2
```

I=3 (3rd iteration):

```
A(3) = B(3) + C(3)    Statement 1

B(4) = A(3)           Statement 2
```

The value of B(2) is used in statement 1 after it is assigned a value in statement 2 of the previous iteration (arrow labeled 1); the value of A(1) is used in statement 2 after it is assigned a value in statement 1 of the same iteration (arrows labeled 2). Due to this interdependence, there is a recurrence cycle.

One way to test for a recurrence cycle is to see if each statement in the loop can execute in a separate loop and maintain the semantics of the program. In the same example, we can put each statement in its own loop as follows:

```
   DO 10 I= 1, N
     A(I) = B(I) + C(I)          Statement 1
10 CONTINUE

   DO 20 I= 1, N
     B(I + 1) = A(I)             Statement 2

20 CONTINUE
```

The value B(2) is used in statement 1 after it is assigned a value in statement 2 in the previous iteration. If we could reverse the order of these statements, they could execute in parallel since all values of B would be assigned in statement 2 before being used in statement 1.

Sometimes, to vectorize, the compiler changes the order of statements if the semantics of the program can be preserved. However, in this example, the order of statements 1 and 2 cannot be reversed because A(1) is assigned in statement 1 and used in statement 2.

The use of the value of B in statement 1 from a previous iteration of statement 2, combined with the requirement that statement 1 execute before statement 2 means that the statements cannot execute in parallel and therefore cannot be vectorized. A recurrence cycle exists. If statement 2 were changed to

```
B(I + 1) = D(I)                  Statement 2
```

then the order of statements could be reversed and they could execute in parallel.

Recurrence cycles prevent vectorization of the statements involved in the cycle. The vectorization report indicates any recurrence cycles. It also indicates any recurrence cycles that contain other recurrence cycles.

Sometimes the vectorizer eliminates a recurrence cycle by using scalar expansion or some other operation. In such cases, no mention of recurrence is made in the vectorization report.

Other types of statements that cannot be vectorized are listed as a recurrence cycle by the vectorization report, even if these statements are not recurrence cycles of the form shown above. For example, implied DO loops in input/output statements can be interpreted as recurrence cycles.

Self-recurrence is a recurrence cycle that occurs on one statement. For example,

```
      DO J=1,10
         A(J)=A(J-1)*B(J)
   10 CONTINUE
```

The value of A(J-1) is computed in a previous iteration of the loop. Self-recurrence occurs for A; therefore, the loop cannot be vectorized.

Vector Operations

The vectorization component of the compiler vectorizes a program by scanning the program and replacing scalar operations with equivalent vector operations. The vector operations include:

Scattering and gathering

Using the sequence vector

Sum reduction

Scalar expansion

Broadcasting

Intrinsic function promotion

Array sectioning

Loop interchange

Induction variable elimination

Stripmining

Loop collapse

These operations are described in this section. This section is intended for those readers desiring more detailed information of the vector-processing methods used in vectorization. Many of the terms described in this section appear in the vectorization report messages. For more information about the vectorization report, see Producing the Implicit Vectorization Report section later in this chapter.

NOTE

The examples presented in this section are only illustrative. For actual examples of complete vectorized programs, see Producing the Implicit Vectorization Report section later in this chapter.

Scattering and Gathering

The scatter and gather operations are used to vectorize statements that reference noncontiguous elements of an array. The scatter instruction is used when contiguous elements of an array are stored into noncontiguous elements of another array.[1] For example:

```
      DIMENSION A(3,3), B(3)
      DO 10 I=1,3
          DO 20 J=1,3
               A(I,J)=B(J)
   20     CONTINUE
   10 CONTINUE
```

In this example, every element of B is assigned to every third element of A in the inner DO loop. The assignment statement assigns elements of array B to the noncontiguous elements of array A. Internally, for the inner loop at I=1:

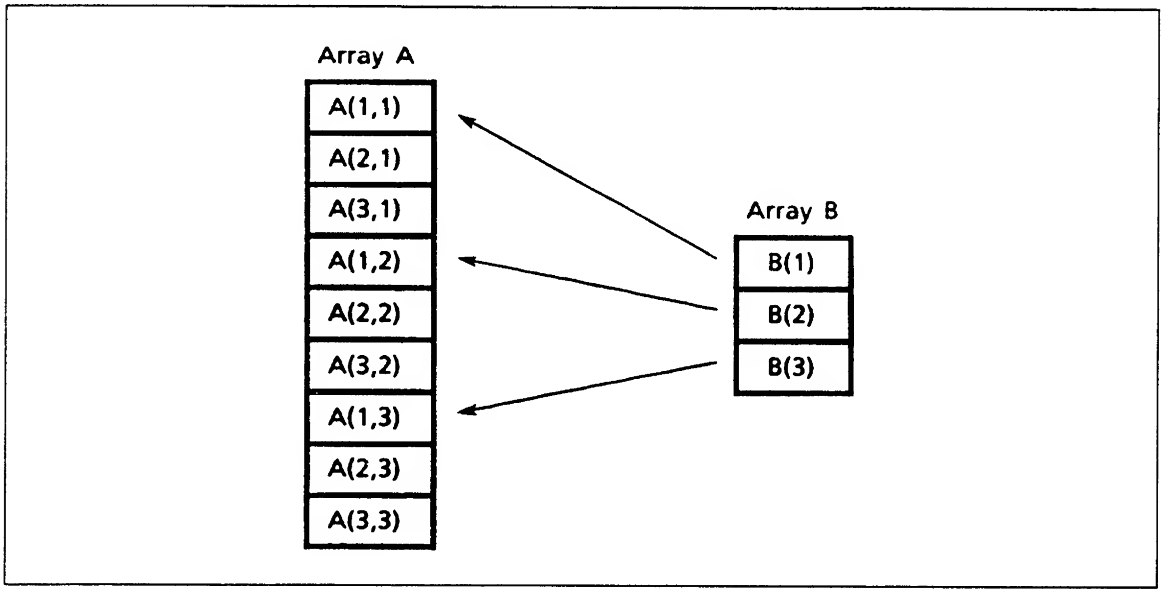


Figure 13-1. The Scatter Instruction

1. Although the iteration count of the DO loop is 3 in this example, it must be greater than 5 for vector processing to occur.

The gather instruction is used to gather noncontiguous elements of an array into an internal temporary vector.[2] For example:

```
      DO 10 I=1,3
          DO 20 J=1,3
               B(J)=B(J) + A(I,J)
20        CONTINUE
10    CONTINUE
```

Internally, for the inner loop at I=1:

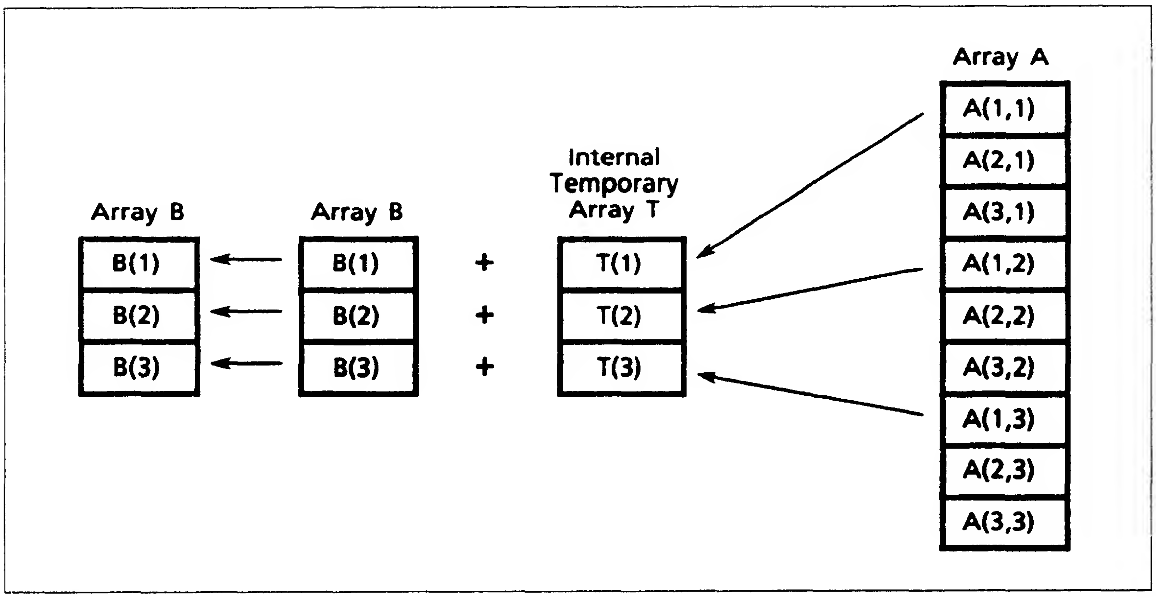


Figure 13-2. The Gather Instruction

2. Although the iteration count of the DO loop is 3 in this example, it must be greater than 5 for vector processing to occur.

The Sequence Vector

Operations involving the assignment of a sequence of integers are vectorized using the sequence vector. The sequence vector is an internal vector containing the integers 1 through N, where N is the largest (or smallest, if negative values are being assigned) integer needed. For example:

```
      DO 10 I=1,500
         A(I)=I
   10 CONTINUE
```

In this example, the integers 1 through 500 are assigned to corresponding array elements. Internally:

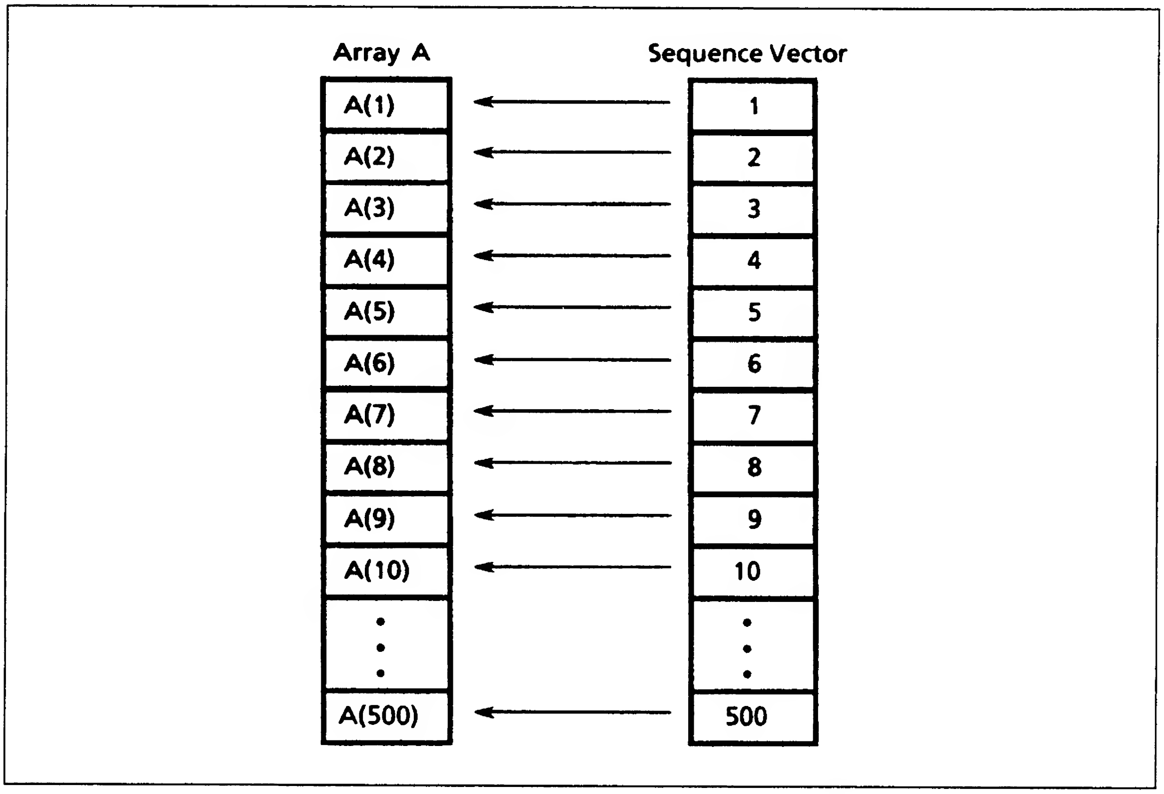


Figure 13-3. The Sequence Vector

You cannot access the sequence vector directly; the compiler generates the sequence vector for internal use only.

Sum Reduction

Some DO loops are vectorized through an operation known as sum reduction. Sum reduction is commonly used when vector elements are summed into a scalar variable. Sum reduction uses the SUM hardware instruction to sum all elements of the vectors, stores the result in a temporary array, and then assigns the result to a scalar variable. Sum reduction works only on type real data. For example:

```
      REAL  B(100), C(100)
      A=0.0
      DO 10 I=1,100
          A=A+B(I)
   10 CONTINUE
```

The sum of all elements of B are stored in an internal temporary value (TEMP), which is added to A and stored into A:

```
TEMP = SUM(B(1:100))

A = A + TEMP
```

Figure 13-4. Sum Reduction

The hardware instruction SUM computes the sum of all elements in TEMP; the sum is added to and assigned to the scalar variable A; the process is then complete:

NOTE

Using the hardware instruction SUM can sometimes generate a different result from the scalar execution of the summation due to a different order of summing the elements (they are summed in parallel). To prevent the use of the hardware instruction SUM, specify EXPRESSION_EVALUATION=MAINTAIN_PRECISION on the VECTOR_FORTRAN command.

Scalar Expansion

Scalar expansion is a technique used in the vectorization of expressions involving scalar values. The vectorizer replaces a scalar with an internal temporary array. This is done so that each iteration through the loop can reference a different memory location, thereby allowing vector processing. For example:

```
      DO 5 I=1,10
           A=B(I)+R(I)        Statement 1
           C(I)=A+D(I)        Statement 2
   5  CONTINUE
```

The vectorizer performs the assignment in statement 1 using a scalar expansion of A and the values in B and R. (Statement 2 is included because the vectorizer does not perform scalar expansion unless the scalar value is needed in a subsequent statement.) Internally, for statement 1 only:

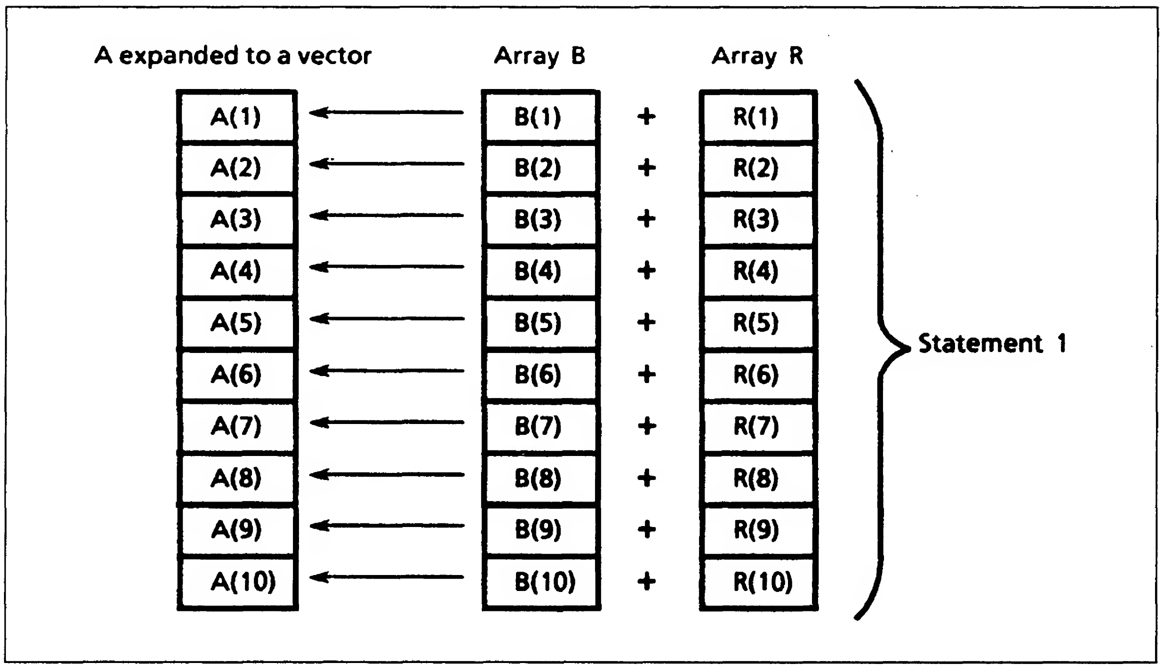


Figure 13-5. Scalar Expansion

When the loop terminates, the value in A(10) is the value assigned to the scalar A; the values of A(1) through A(9) are not used in calculating the value of A at loop termination.

Broadcasting

Broadcasting is the treatment of a scalar value as a temporary vector. The purpose of broadcasting is to perform vector operations when one of the operands is a scalar value. For example:

```
      B=1.0
      DO 10 I=1,10
         A(I)=B            Statement 1
   10 CONTINUE
```

In statement 1, the value of B (in this case 1.0) is assigned to an element of array A on each iteration of the loop. The statement is vectorized using the broadcast operation:

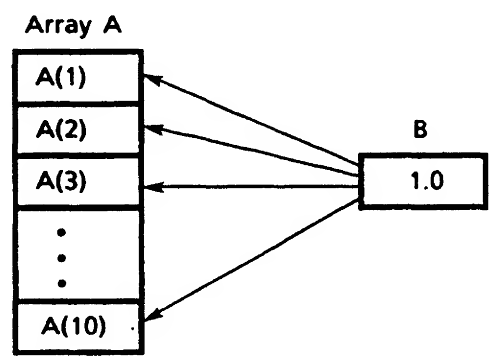


Figure 13-6. Broadcasting

The vectorization report does not indicate when broadcasting occurs.

Intrinsic Function Promotion

The vectorizer replaces certain intrinsic function references and expressions involving exponentiation with equivalent vector versions. This operation is called intrinsic promotion.

For example:

```
      DO 10 I=1,10
         A(I)=SIN (B(I))
   10 CONTINUE
```

The vector version of the SIN intrinsic can process a vector of values:

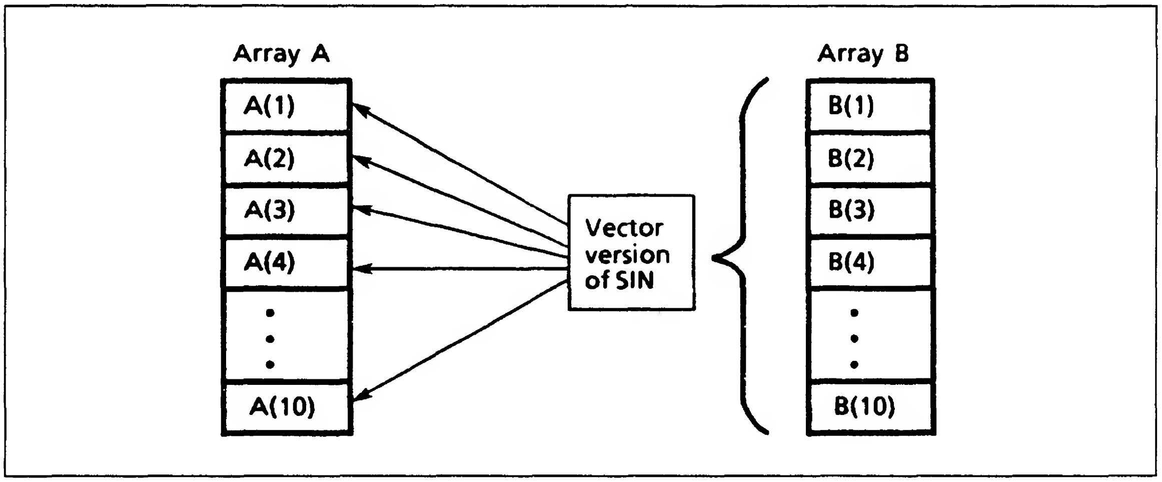


Figure 13-7. Intrinsic Promotion

The vectorization report indicates which intrinsics are replaced by vector versions. The vectorizable intrinsic functions include all of the mathematical elemental intrinsic functions and most of the array-processing intrinsic functions. Intrinsic functions are described in chapter 9.

The math routines called by exponentiation operations also have vector equivalents. For example, the following statement can be vectorized:

A(I)=X(I)**5

The vector entry point of the Math Library routine is used.

Array Sectioning

Array sectioning is the process of transforming array element references into an array section reference. Vector operations are then performed on the array section. For example:

```
      DIMENSION A(50, 50)
      DO 15 J=1,10
          DO 10 I=1,30,3           !
             A(I, J)=4+B(J)        ! Inner loop
10        CONTINUE                 !
15    CONTINUE
```

Array A is sectioned (processed as a vector) along index I. You can perform array sectioning explicitly using array section notation. For example, the three statements from the inner loop could be written in array section notation as follows:

```
A(1:30:3, J) = 4 + B(J)
```

The vectorization report indicates which array element references are transformed into array section references.

Loop Interchange

In some nested DO loops, the index of an outer DO loop can be sectioned, but the index of an inner DO loop cannot. If certain conditions are met, the two loops of a loop nest can be interchanged such that the outer loop becomes the inner loop and vice versa. The following example illustrates this:

```
      DO 10 I=1, N
       DO 10 J=1, I
10      A(I, J)=A(I, J-1)+100.0
```

Index J cannot be sectioned because of the recurrence on J in this statement. However, if the loops are interchanged, index I can be sectioned. Internally:

```
      DO 10 J=1, N
       DO 10 I=J, N
        A(I, J)=A(I, J-1)+100.0
   10 CONTINUE
```

After sectioning:

```
      DO 10 J=1, N
10    A(J:N, J)=A(J:N, J-1)+100.0
```

Note that the initial and terminal parameter of the two loops must be changed to maintain the semantics of the loop in this example. The vectorizer changes loop limits if necessary when performing loop interchange.

Induction Variable Elimination

Induction variable elimination attempts to replace variables that are functions of a DO variable with an expression involving the DO variable. This increases the number of occurrences of the DO variable in the loop and allows more statements to be sectioned.

Example:

```
      DO 10 I = 1, 10
         K = I + 5
         A(K) = B(K) + 10.0
   10 CONTINUE
```

Internally:

```
      DO 10 I = 1, 10
         K = I + 5
         A(I + 5) = B(I + 5) + 10.0
   10 CONTINUE
```

Stripmining

Stripmining is used by the vectorizer to vectorize loops where the iteration count either is unknown (as in an assumed-size array) or exceeds the hardware vector length of 512. Dummy argument arrays that are dimensioned to size 1 are treated as assumed-size arrays and are candidates for stripmining in the vectorization process. For example, the lengths of the following vectors exceed 512:

```
      DO 10 I=1,1000
         A(I)=B(I)
   10 CONTINUE
```

Internally, two separate vector operations are performed:

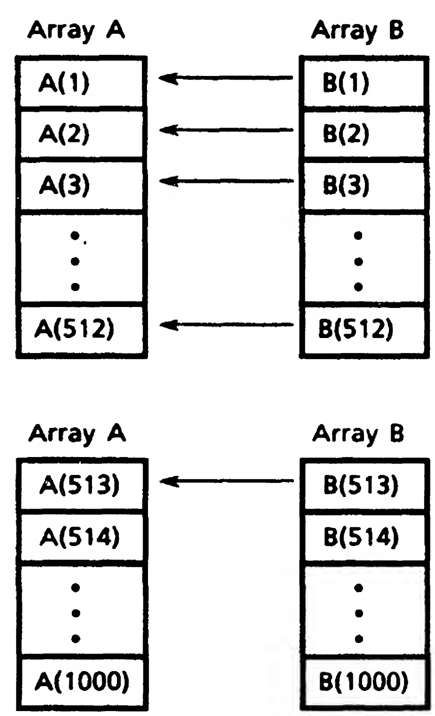


Figure 13-8. Stripmining

You should use the C$ ASSUME(MAXITER) directive when the iteration count of a DO loop is less than 512 and the iteration count is not known at compile time.

For example:

```
      DIMENSION A(1000), B(1000)
      C$ ASSUME(MAXITER=100)
      DO 10 I=1, N
       A(I)=B(I)
   10 CONTINUE
```

This example assumes N is equal to 100. The assignment A(I)=B(I) occurs without stripmining.

You must ensure that the actual iteration count of a DO loop does not exceed the iteration count specified on the C$ ASSUME(MAXITER) directive.

Loop Collapse

When a multidimensional array is vectorized, active contiguous dimensions are collapsed to a single dimension. Active dimensions are dimensions that are referenced in the DO loop. Loop collapse occurs only when the array section bounds (which are the same as the loop bounds) span the array dimension. For example:

```
      DIMENSION A(5, 15), B(5, 15)
      DO 10 I=1, 10
         DO 20 J=1, 5
           A(J, I)=B(J, I)+5.0
20       CONTINUE
10    CONTINUE
```

Since the inner DO loop iterates five times, and arrays A and B are of size 5 in the first dimension, all elements of the arrays being referenced are contiguous, and the array section bounds span the first array dimension. The two dimensions of A and B can be joined, or collapsed, into one single dimension of size 50:

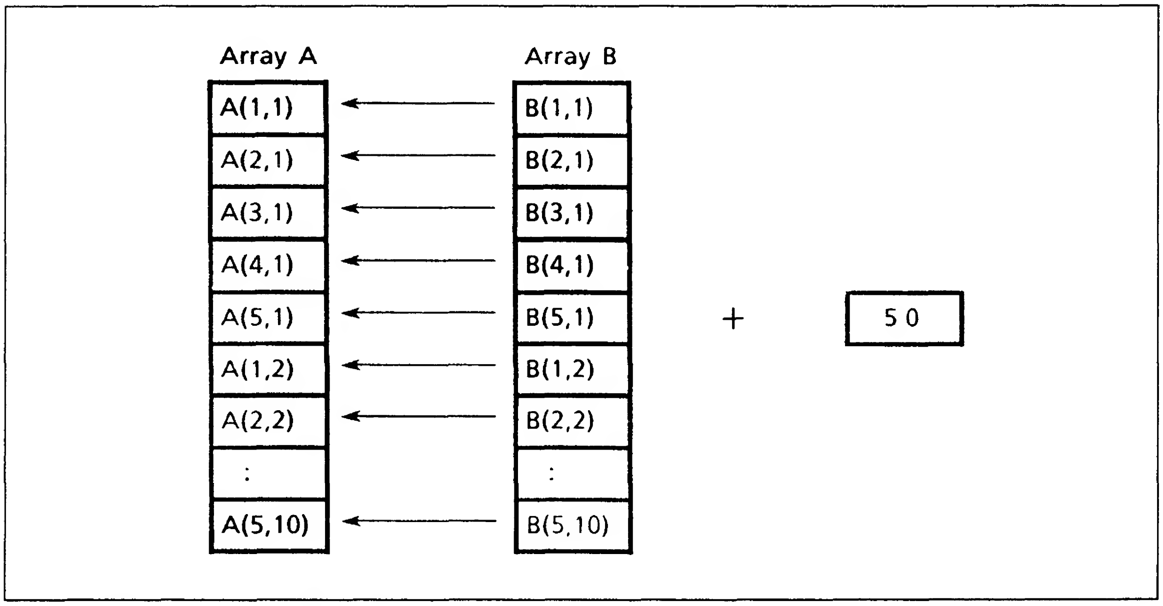


Figure 13-9. Loop Collapse

In this example, both loops are vectorized, and the entire double loop nest becomes a single vector operation. (However, if the entire double loop nest has an iteration count greater than 512, or unknown at compile time, it will be stripmined.)

Controlling Implicit Vectorization with C$ Directives

Vectorization control directives are used to control implicit vectorization of lines in your program. There are three directives:

C$ ASSUME(NORECUR)
C$ ASSUME(MAXITER=c)
C$ CONTROL(p)

A C$ directive is identified by the letter C in position 1 together with the character $ in position 2. Such a line is interpreted as a comment if the COMPILATION_DIRECTIVES parameter is not selected on the VECTOR_FORTRAN command.

C$ ASSUME(NORECUR)

The C$ ASSUME(NORECUR) directive indicates that no recurrence cycle exists in the immediate following statement. This directive has the form:

C$ ASSUME(NORECUR)

The C$ ASSUME(NORECUR) directive can only appear within a DO loop.

Example:

```
      DO 5 I = 1, N
C$    ASSUME(NORECUR)
      A(I) = A(I + K) + 1.0
    5 CONTINUE
```

The C$ ASSUME directive eliminates the possibility of the variable K causing a recurrence cycle; therefore, the user is sure that K is always greater than or equal to 0.

C$ ASSUME(MAXITER=c)

The C$ ASSUME directive indicates the maximum iteration count of a DO loop. The C$ ASSUME directive has the form:

C$ ASSUME(MAXITER=c)

where c is an integer constant or symbolic constant.

The C$ ASSUME(MAXITER=c) statement must appear directly before a DO statement; otherwise it is ignored. This information prevents unnecessary stripmining instructions and the vectorization of small (less than 5 iterations) loops.

Example:

```
C$    ASSUME(MAXITER = 100)
      DO 5 I = 1, M + N
         A(I) = B(I) + 1.0
    5 CONTINUE
```

You should also use the C$ ASSUME(MAXITER) directive for loops that iterate less than five times (but whose iteration count is not known at compile time) to avoid preparation for stripmining.

C$ CONTROL(p)

The C$ CONTROL directive indicates which source lines of a program are ignored in vectorization. The C$ CONTROL directive has the form:

C$ CONTROL(p)

p is one of the following:

VEC

Enables vectorization for subsequent lines. The compiler attempts to vectorize all subsequent lines until either a C$ CONTROL(NOVEC) directive appears or the end of the program unit is encountered.

NOVEC

Disables vectorization for subsequent lines.

Example:

```
      DO 5 I = 1, N
      A(I) = B(I) + C(I)
C$    CONTROL(NOVEC)                ! <--- Vectorization
       DO 7 J = 1, M                !      is disabled
        R(I,J) = 2.0 * S(I,J)       !      for these
   7   CONTINUE                     !      lines.
C$    CONTROL(VEC)                  ! <--- Vectorization
      D(I) = A(I) - 1.0             !      is enabled for
   5  CONTINUE                      !      these and
                                    !      following lines.
```

The inner DO loop is not a candidate for vectorization. The first and last statements of the outer DO loop are candidates for vectorization.

If a DO statement is written within the range of a C$ CONTROL(NOVEC) directive, the control variable for the DO loop is not sectioned even if a subsequent C$ CONTROL (VEC) directive appears within the DO loop. Any inner loops nested within the DO loop are candidates for vectorization if vectorization has been enabled by the C$ CONTROL(VEC) directive.

Example:

```
C$    CONTROL(NOVEC)
      DO 20 I=1, 25
C$    CONTROL(VEC)
      DO 30 J=1, 10
      A(I,J) = B(I, J) + C(I,J)
 30   CONTINUE
      A(I,1) = B(I,1)
 20   CONTINUE
```

Vectorization is disabled for index I in the inner DO loop and enabled for index J.

Producing the Implicit Vectorization Report

The FORTRAN compiler provides an optional vectorization report which consists of messages that indicate the degree of implicit vectorization performed on your object program. You can select either full or brief mode messages, the former providing more detail.

The vectorization report messages appear on the listing file along with any WARNING or TRIVIAL compilation diagnostics. The listing file is the file specified by the LIST parameter on the VECTOR_FORTRAN command. Report messages are distinguished by the characters CV before the message number and the word REPORT along the left margin of the listing file. No report messages appear if FATAL compilation errors are present.

To generate the vectorization report, specify REPORT_OPTIONS=BRIEF or REPORT_OPTIONS=FULL on the VECTOR_FORTRAN command. The default selection for REPORT_OPTIONS is NONE. The two options on the REPORT_OPTIONS parameter indicate brief or full mode messages, respectively. Brief mode is usually sufficient to convey the vectorization performed on your program. The brief mode messages are a subset of the full mode messages. Full mode messages indicate constructs that were not vectorized. Full mode also gives a summary of all recurrence cycles found in the program.

Remember to specify both OPTIMIZATION_LEVEL=HIGH and VECTORIZATION_LEVEL=HIGH, as well as the REPORT_OPTIONS parameter when compiling your program.

The following examples show the results of vectorization as reported by the vectorization report.

Example (brief mode):

```
      1       PROGRAM T
      2       REAL A(8),R
      3       R=0
      4       DO 10 I=1,8
      5         R=R+(A(I)+8)
 REPORT  ------>  CV 4210  SUM reduction used to vectorize line 5.
      6    10 CONTINUE
      7       PRINT *,'R=', R
      8       END
```

The vectorization message of this example indicates that line 5 was fully vectorized using sum reduction.

By compiling the same source program with RO=FULL, an additional vectorization message is generated. It indicates the array and dimension used in the reduction:

```
     1         PROGRAM T
     2         REAL A(8),R
     3         R=0
     4         DO 10 I=1,8
REPORT   ------>  CV 4226  A sectioned on index I, on line(s) 5.
     5           R=R+(A(I)+8)
REPORT   ------>  CV 4210  SUM reduction used to vectorize line 5.
     6     10  CONTINUE
     7         PRINT *,'R=', R
     8         END
```

The vectorization messages of the following example indicate that the assignment statement referencing array C was fully vectorized in both DO loops. The assignment statement referencing array D was not vectorized because it is not used in any subsequent statement in the program. (It is actually removed from the object code by the optimizer.) This example was compiled with REPORT_OPTIONS=BRIEF:

```
     1         PROGRAM TEST27
     2         INTEGER A,B
     3         REAL C,D
     4         DIMENSION C(15),D(15)
     5         A=0
     6         B=0
     7         DO 10 I=1, 15
REPORT   ------>  CV 4228  Line(s) 8 vectorized on index I.
     8         C(I)=0.0
     9         D(I)=0.0
    10  10       CONTINUE
    11          DO 30 I=1, 15
REPORT   ------>  CV 4228  Line(s) 12 vectorized on index I.
    12         C(I)=C(I)+I
    13   30      CONTINUE
    14         END
```

By compiling the same program with RO=FULL, more information about how the DO loops were vectorized is given:

```
         1       PROGRAM TEST27
         2       INTEGER A,B
         3       REAL C,D
         4       DIMENSION C(15),D(15)
         5       A=0
         6       B=0
         7       DO 10 I=1, 15
REPORT    ------> CV 4226  C sectioned on index I, on line(s) 8.
         8       C(I)=0.0
         9       D(I)=0.0
        10  10     CONTINUE
        11        DO 30 I=1, 15
REPORT    ------> CV 4226  C sectioned on index I, on line(s) 12.
        12       C(I)=C(I)+I
REPORT    ------> CV 4222  Sequence vector generated.
        13   30     CONTINUE
        14       END
```

Array C was sectioned (to allow for vector processing) on index I (the DO variable of the surrounding loop) in both DO loops. The sequence vector was also used in the vectorization of statement 12 since it involved the assignment of integer values.

The following example shows a program compiled with RO=BRIEF:

```
         1       PROGRAM TEST28
         2       INTEGER A,B
         3       REAL C,D
         4       DIMENSION C(530),D(530)
         5       A=0
         6       B=0
         7       DO 10 I=1, 530
REPORT   ------>  CV 4228  Line(s) 8-9 vectorized on index I.
         8         C(I)=0.0
REPORT   ------>  CV 4328  C stripmined.
         9         D(I)=0.0
REPORT   ------>  CV 4328  D stripmined.
        10  10   CONTINUE
        11       DO 20 I=1, 530
REPORT   ------>  CV 4228  Line(s) 12 vectorized on index I.
        12       C(I)=SIN(C(I))
REPORT   ------>  CV 4328  C stripmined.
REPORT   ------>  CV 4220  Intrinsic sin replaced by vector version.
        13  20   CONTINUE
        14       DO 30 I=1, 530
REPORT   ------>  CV 4224  Line(s) 15 contains self-recurrence for the
                           loop on line 14.
        15       C(I)=SIN(C(I-1))
        16  30   CONTINUE
        17       PRINT *, C, D
        18       END
```

The vectorization messages indicate the following:

- Message CV 4228 indicates that the assignment statement involving array C and array D were fully vectorized.
- Message CV 4228 shows that the statement on line 12 was vectorized successfully.
- Message CV 4220 indicates that the vector version of the intrinsic function SIN was used.
- Message CV 4224 indicates the self-recurrence on line 15. The self-recurrence occurs because C(I-1) references an array element that was stored into in a previous iteration of the loop.
- Message CV 4328 indicates that the vectors were stripmined.

Implicit Vectorization Messages

The following table lists all report messages and their associated numbers, whether they appear in brief or full mode, and a brief description. The symbol +P indicates a value that is inserted when the message is generated.

Message Number: 4101

Message:	Loop not vectorized; contains noninteger DO variable.
Mode:	Brief, Full
Description:	The DO loop cannot be vectorized because it contains a DO variable that is not of type integer. No statements in the DO loop are vectorized.

Message Number: 4102

Message:	Loop not vectorized; contains branch out of loop.
Mode:	Brief, Full
Description:	The DO loop cannot be vectorized because it contains a potential branch out of the DO loop. No statements in the DO loop are vectorized.

Message Number: 4103

Message:	Loop not vectorized; C$ CONTROL (NOVEC) directive received.
Mode:	Brief, Full
Description:	The DO control variable of this DO loop is not sectioned for any statements within the DO loop.

Message Number: 4104

Message:	Loop not vectorized; iteration count less than or equal to 5.
Mode:	Brief, Full
Description:	Loops which iterate five or less times are not vectorized.

Message Number: 4120

Message:	Vectorization impediment on line(s) +P.
Mode:	Brief, Full
Description:	The program construct on the indicated line(s) cannot be vectorized. However, other statements in the DO loop might be vectorized. If +P is a range of lines, the impediment extends throughout the range of lines. Statements that cannot be vectorized are usually ones that are executed an indeterminate number of times; for example, statements controlled by an arithmetic IF cannot be vectorized.

Message Number: 4122

Message: Recurrence cycle +P contains cycle(s) +P.

Mode: Full

Description: The indicated recurrence cycle contains another recurrence cycle or cycles. Use the recurrence cycle number to locate the recurrence cycle and its associated lines in message 4124. Statements within recurrence cycles cannot be vectorized.

Message Number: 4124

Message: Recurrence cycle +P contains line(s) +P.

Mode: Full

Description: The indicated recurrence cycle contains the indicated line or lines. A recurrence cycle cannot be vectorized.

Message Number: 4126

Message: +P expanded as array +P for loop(s) on line +P.

Mode: Brief, Full

Description: Scalar expansion was used on the indicated scalar on the indicated line. The scalar was expanded into a temporary internal array. The name may be helpful for debugging purposes.

Internal temporary arrays or scalars are created to hold temporary values. These internal locations are assigned names by the compiler; the names may be helpful for debugging purposes. Some examples of internal names that appear on the vectorization report are:

Internal_VCG_array.1
A_ES.20
B_DARG.2
VGTHR.1

Message Number: 4200

Message: Nonvectorizable external call or function reference.

Mode: Full

Description: A statement causing a reference to an external routine cannot be vectorized. This message does not appear with input/output statements, although those statements cause references to external routines and cannot be vectorized.

Message Number: 4201

Message: Character type cannot be vectorized.

Mode: Full

Description: A statement containing character operands cannot be vectorized.

Message Number: 4210

Message: SUM reduction used to vectorize line +P.

Mode: Brief, Full

Description: Sum reduction was performed on the statement on the indicated line. Sum reduction is used for type real data only.

Message Number: 4220

Message: Intrinsic +P replaced by vector version.

Mode: Brief, Full

Description: The indicated intrinsic function was replaced by an equivalent but faster vector version.

Message Number: 4222

Message: Sequence vector generated.

Mode: Full

Descripton: The sequence vector was generated and used to vectorize the assignment of integer values.

Message Number: 4224

Message: Line(s) +P contains self-recurrence for the loop on line +P.

Mode: Brief, Full

Description: Self-recurrence exists for the indicated variable on the indicated line. Statements containing self-recurrence cannot be vectorized.

Message Number: 4226

Message: +P sectioned on index +P, on line(s) +P.

Mode: Full

Description: The indicated array was sectioned on the indicated index (DO variable).

Message Number: 4228

Message: Line(s) +P vectorized on index +P.

Mode: Brief

Description: The indicated array was vectorized (sectioned) along the indicated index (DO variable).

Message Number: 4230

Message: Line(s) +P not vectorized; C$ CONTROL (NOVEC) directive received.

Mode: Brief, Full

Description: The indicated line(s) were not vectorized because a C$ CONTROL (NOVEC) directive was specified.

Message Number: 4240

Message: Induction variable +P replaced with expression containing DO control variable +P on line +P.

Mode: Full

Description: The induction variable has been replaced with an expression which contains the DO variable.

Message Number: 4310

Message: +P gathered into +P.

Mode: Full

Description: The hardware function GATHER was used to process elements of the indicated array into the indicated internal variable. Noncontiguous elements are gathered to contiguous locations before vector processing can occur. The name of the internal variable may be useful when reading object code listings.

Internal temporary arrays or scalars are created to hold temporary values. These internal locations are assigned names by the compiler; the names may be helpful for debugging purposes. Some examples of internal names that appear on the vectorization report are:

Internal_VCG_array.1
A_ES.20
B_DARG.2
VGTHR.1

Message Number: 4312

Message: +P scattered into +P.

Mode: Full

Description: The hardware function SCATTER was used to assign the elements of one array to another. Elements from an internal array are scattered to noncontiguous locations in an array after vector processing has occurred. The array names may not be names from your program. Internal temporary arrays or scalars are sometimes created to hold temporary values. These internal locations are assigned names by the compiler; the names may be helpful for debugging purposes. Some examples of internal names that appear on the vectorization report are:

Internal_VCG_array.1
A_ES.20
B_DARG.2
VGTHR.1

Message Number: 4320

Message: +P collapsed on dimension(s) +P.

Mode: Full

Description: The vectorizer recognized contiguous elements of the indicated n-dimensional array and collapsed the indicated dimensions.

Message Number: 4322

Message: Scalar loop generated for dimension(s) +P of +P.

Mode: Full

Description: The indicated dimensions of the indicated array were processed with scalar instructions. Scalar processing is performed on only one dimension in a loop when:

- Loop collapse is not possible
- The iteration count of the loop is less than 6
- The data type of the array is not supported in vector processing

Message Number: 4324

Message: Scalar loops generated for all dimensions of +P.

Mode: Full

Description: All dimensions of the indicated array were processed with scalar instructions.

Message Number: 4326

Message: Line +P partially vectorized.

Mode: Brief

Description: The statement on the indicated line was partially vectorized. That is, some vector instructions were issued for the indicated statement, but some scalar instructions were still necessary.

Message Number: 4328

Message: +P stripmined.

Mode: Brief, Full

Description: The indicated array was stripmined because its size was unknown or greater than 512. The array name may not be a symbolic name from your program. Internal temporary arrays or scalars are sometimes created to hold temporary values. These internal locations are assigned names by the compiler; the names may be helpful for debugging purposes. Some examples of internal names that appear on the vectorization report are:

```
Internal_VCG_array.1
A_ES.20
B_DARG.2
VGTHR.1
```

Examples 14

Program PASCAL 14-1
Program CORR 14-3
Program COMPSAL 14-6
Subroutine COUNTC 14-8
Mass Storage Input/Output Examples 14-10
Program SCLCALL 14-12

Examples 14

This chapter shows complete executable programs along with examples of input, output, and terminal dialog where appropriate. Note that for examples showing actual terminal dialog, the dialog displayed at your terminal might vary slightly depending on the characteristics of the terminal.

Program PASCAL

Program PASCAL, shown in figure 14-1, generates a Pascal triangle. The program illustrates the use of DO loops, including nested DO loops and a loop with a negative index parameter.

```
      PROGRAM PASCAL
C
C     THIS PROGRAM PRODUCES A PASCAL TRIANGLE.
C
      INTEGER LROW(15)
      DO 10 I=1,15
         LROW(I) = 1
 10      CONTINUE
      PRINT '(" PASCAL TRIANGLE"//1X,I5,/1X,2I5)',
     +LROW(15), LROW(14), LROW(15)
      DO 20 J=14,2,-1
         DO 15 K=J,14
            LROW(K) = LROW(K) + LROW(K+1)
 15   CONTINUE
         PRINT '(1X,15I5)', (LROW(M), M=J-1,15)
 20   CONTINUE
      END
```

Figure 14-1. Program PASCAL

The INTEGER statement declares a 15-word array to be used to contain the elements of a row of the triangle. The first DO loop initializes all elements of the array to 1. The first PRINT statement prints the first two rows of the triangle; the format specification uses the slash descriptor to print multiple lines.

The nested DO loops calculate and print the remaining rows of the triangle. The value of the first and last element of each row is one. Each remaining element is calculated by adding the corresponding element in the preceding row to its preceding element. (For example, the third element of row three is calculated by adding the second and third elements of row 2, and so forth.)

The triangle produced by program PASCAL is shown in figure 14-2.

```
PASCAL TRIANGLE

   1
   1    1
   1    2    1
   1    3    3    1
   1    4    6    4    1
   1    5   10   10    5    1
   1    6   15   20   15    6    1
   1    7   21   35   35   21    7    1
   1    8   28   56   70   56   28    8    1
   1    9   36   84  126  126   84   36    9    1
   1   10   45  120  210  252  210  120   45   10    1
   1   11   55  165  330  462  462  330  165   55   11    1
   1   12   66  220  495  792  924  792  495  220   66   12    1
   1   13   78  286  715 1287 1716 1716 1287  715  286   78   13    1
   1   14   91  364 1001 2002 3003 3432 3003 2002 1001  364   91   14    1
```

Figure 14-2. Program PASCAL Output

Program CORR

Program CORR, shown in figure 14-3, reads two sets of integers from the terminal and calculates a correlation coefficient. Program CORR illustrates the following features:

PARAMETER statement

Interactive input and output

The correlation coefficient measures the correlation between two sets of numbers. A coefficient with a value close to 1 indicates close correlation.

```
      PROGRAM CORR
C
C     ASSIGN SYMBOLIC NAME N TO CONSTANT 10.
C
      PARAMETER (N=10)
C
      INTEGER SUMJ, SUMK, SUMJK, SUMJSQ, SUMKSQ, J(N), K(N)
      REAL NUM
C
C     READ NUMBERS TO BE CORRELATED.
C
   10 CONTINUE
      PRINT*, 'ENTER FIRST SET OF ', N, ' NUMBERS'
      READ*, J
      IF (J(1) .EQ. 9999) STOP
      PRINT*, 'ENTER SECOND SET OF ', N, ' NUMBERS'
      READ*, K
C
C     INITIALIZATION.
C
      SUMJ = 0
      SUMK = 0
      SUMJSQ = 0
      SUMKSQ = 0
      SUMJK = 0
C
C     CALCULATE CORRELATION COEFFICIENT.
C
      DO 20 I = 1,N
         SUMJ = SUMJ + J(I)
         SUMK = SUMK + K(I)
         SUMJSQ = SUMJSQ + J(I)**2
         SUMKSQ = SUMKSQ + K(I)**2
         SUMJK = SUMJK + J(I) * K(I)
  20       CONTINUE
C
      NUM = REAL(N * SUMJK - SUMJ * SUMK)
      A = REAL(N * SUMJSQ - SUMJ**2)
      B = REAL(N * SUMKSQ - SUMK**2)
      DENOM = SQRT(A) * SQRT(B)
      R = NUM/DENOM
      PRINT 100, R
 100  FORMAT (' CORRELATION COEFFICIENT = ',F6.2,//)
      GO TO 10
      END
```

Figure 14-3. Program CORR

Program CORR reads two sets of numbers from the terminal. If the first number of the first set is 9999, the program immediately stops; otherwise, the program performs the calculation and branches to the beginning to request another set of input values. The program is written to process sets containing 10 numbers each. The statement PARAMETER (N=10) assigns the name N to the constant 10. This symbolic constant is used in the DIMENSION statement, the DO statement, and in the statements that calculate the intermediate values NUM, A, and B. The program can be modified to calculate a result for a different number of values simply by changing the PARAMETER statement.

The two PRINT statements at the beginning of the program provide an informative prompt for input. Because the UNIT specifier is omitted from the succeeding READ statements, unit INPUT is implied. When the READ statements are executed, the system prints a question mark, and execution stops until the user types a set of input values.

An example of terminal dialog for program CORR, showing input and output, is shown in figure 14-4.

```
/lgo
ENTER FIRST SET OF 10 NUMBERS
? 1 2 3 4 5 6 7 8 9 10
ENTER SECOND SET OF 10 NUMBERS
? 1 2 3 4 5 6 7 8 9 10
CORRELATION COEFFICIENT =   1.00


ENTER FIRST SET OF 10 NUMBERS
? 5 96 127 0 3 25 84 16 22 50
ENTER SECOND SET OF 10 NUMBERS
? 0 0 4 18 9 56 32 0 0 10
CORRELATION COEFFICIENT =   -.05


ENTER FIRST SET OF 10 NUMBERS
? 3 4 5 3 4 5 3 4 5 3
ENTER SECOND SET OF 10 NUMBERS
? 3 4 5 3 4 5 3 2 1 0
CORRELATION COEFFICIENT =    .39


ENTER FIRST SET OF 10 NUMBERS
? 9999 0 0 0 0 0 0 0 0 0
```

Figure 14-4. Program CORR Output

Program COMPSAL

Program COMPSAL, shown in figure 14-5, calculates salaries from data input at the terminal. This program illustrates interactive input and output, and the use of block IF structures.

```
      PROGRAM COMPSAL
      CHARACTER NAME*20
      INTEGER AGE
   10 CONTINUE
C
C      PROMPT FOR INPUT.
C
       PRINT*, 'TYPE NAME, AGE, AND WAGES'
C
C      READ INPUT VALUES.  UNIT=* READS FROM FILE INPUT.
C      FMT=* SPECIFIES LIST DIRECTED INPUT.
C
       READ (UNIT=*,FMT=*) NAME, AGE, WAGES
C
C      CALCULATE SALARY, DEPENDING ON VALUE OF AGE.
C
       IF (AGE .GT. 65) THEN
          SALARY = WAGES*0.7
       ELSE IF (AGE .GT. 60) THEN
          SALARY = WAGES*0.6
       ELSE IF (AGE .LT. 18) THEN
          SALARY = WAGES*0.52
       ELSE
          SALARY = WAGES
       ENDIF
C
       PRINT 100, SALARY
  100  FORMAT (/, ' SALARY IS $', F8.2, //)
C
C      TEST FOR LAST INPUT NAME.
C
       IF (NAME(1:1) .EQ. '/') STOP
C
C      BRANCH BACK TO READ ANOTHER LINE.
C
       GO TO 10
       END
```

Figure 14-5. Program COMPSAL

Program COMPSAL reads a line from the terminal, containing values for NAME, AGE, and WAGES. The UNIT=* specifier in the READ statement causes the program to read from unit INPUT, which is connected to the terminal.

The program uses a block IF structure to test the value of WAGES and calculate a value for SALARY depending on the result of the test. The program then tests for a slash in the first position of the input line. If a slash is found, execution stops; if no slash is found, control transfers to the beginning to read another input line.

A sample terminal dialog for program COMPSAL is shown in figure 14-6.

```
/lgo
TYPE NAME, AGE, AND WAGES
? 'smith' 70 12000

SALARY IS $ 8400.00


TYPE NAME, AGE, AND WAGES
? 'jones' 33 8000

SALARY IS $ 8000.00


TYPE NAME, AGE, AND WAGES
? '/hansen' 16 1000

SALARY IS $  520.00
```

Figure 14-6. Sample Terminal Dialog for Program COMPSAL

Subroutine COUNTC

Subroutine COUNTC, shown in figure 14-7, counts the number of occurrences of a specified character in the input line. This program illustrates the use of character substrings and a variable length CHARACTER specification.

```
      PROGRAM MAIN
      CHARACTER LINE*80, CHAR*1
      PRINT*, ' TYPE A LINE'
      READ*, LINE
      PRINT*, ' TYPE A CHARACTER'
      READ*, CHAR
      CALL COUNTC (LINE, CHAR, NCHAR)
      PRINT 111, CHAR, NCHAR
 111  FORMAT (/, ' CHARACTER ', A1, ' OCCURRED ', I2, ' TIMES ', //)
      END

      SUBROUTINE COUNTC (A, CH, N)
C*
C* DECLARE A TO HAVE THE LENGTH USED IN THE CALL.
C*
      CHARACTER A*(*), CH*1
      N = 0
C*
C* TEST EACH CHARACTER IN INPUT LINE.  IF MATCH IS SUCCESSFUL,
C* INCREMENT COUNTER. IF PERIOD, RETURN.
C*
      DO 10 I = 1,LEN(A)
      IF (A(I:I) .EQ. CH) THEN
          N = N + 1
      ELSE IF (A(I:I) .EQ. '.') THEN
          RETURN
      ENDIF
 10   CONTINUE
      END
```

Figure 14-7. Subroutine COUNTC

Subroutine COUNTC has three dummy arguments: argument A receives the input character string, CH receives the character to be tested, and N returns the number of occurrences of the character passed in CH. The argument A is declared to have length (*). This means that when COUNTC is called, A will have the length of the string passed through A. Thus, a string of any length can be passed (although the main program accepts no more than 80 characters).

The DO loop contains a block IF structure that tests each character of the input line for the occurrence of the input character. If the input character is detected, a counter is incremented. If the input character is not detected, the ELSE IF statement tests for a period. If a period is found, control returns to the calling program; otherwise, the next character in the line is tested. Each character is tested until either a period is found or the entire string has been tested; control then returns to the calling program. The main program in figure 14-7 reads a character string from the terminal, reads a single character, calls COUNTC, and prints the results. Figure 14-8 shows an example of terminal dialog and resulting output.

```
/lgo
TYPE A LINE
? 'this is the first line.'
TYPE A CHARACTER
? 't'

CHARACTER t OCCURRED  3 TIMES.
```

Figure 14-8. Sample Terminal Dialog for Subroutine COUNTC

Mass Storage Input/Output Examples

Figure 14-9 contains two programs that illustrate random file operations. Program MS1 creates a random file with number index and writes 10 records to the file. Program MS2 performs the following modifications to the file created by program MS1:

- Modifies record 8 and rewrites it at end-of-data.
- Modifies record 6 and rewrites it in place.
- Replaces record 2 with a longer record.
- Adds two new records to the end of the file.

```
      PROGRAM MS1
C
      DIMENSION INDEX(11), DATA(25)
      CALL OPENMS (3, INDEX, 11, 0)
      DO 50 NKEY=1, 10
         :  (Generate record in array DATA)
          CALL WRITMS (3, DATA, 25, NKEY, 0)
  50  CONTINUE
      END

      PROGRAM MS2
C
C     NOTE LARGER INDEX ARRAY TO ACCOMMODATE TWO NEW RECORDS.
C
      DIMENSION INDEX(13), DATA(25), MORDAT(40)
      CALL OPENMS (3, INDEX, 13, 0)
C
C     READ RECORD 8 FROM FILE TAPE3.
      CALL READMS (3, DATA, 25, 8)
         : (Modify array DATA)
C     WRITE MODIFIED ARRAY AS RECORD 8 AT END-OF-INFORMATION.
       CALL WRITMS (3, DATA, 25, 8, 0)
C
C     READ RECORD 6
       CALL READMS (3, DATA, 25, 6)
         :   (Modify array DATA)
C     REWRITE MODIFIED ARRAY IN PLACE AS RECORD 6
       CALL WRITMS (3, DATA, 25, 6,-1)
C
C     READ RECORD 2 INTO LONGER ARRAY MORDAT
       CALL READMS (3, MORDAT, 25, 2)
         : (Add 15 new words to array MORDAT)
C     SPECIFY IN-PLACE REWRITE OF RECORD 2.
C     HOWEVER, BECAUSE NEW RECORD IS LONGER THAN
C     OLD RECORD, THE NEW RECORD IS WRITTEN AT
C     END-OF-DATA.
       CALL WRITMS (3, MORDAT, 40, 2,-1)
       END
```

Figure 14-9. Programs MS1 and MS2

The example in figure 14-10 creates a subindexed file with a number index. Program MS4 creates four subindexes and writes nine data records for each subindex, for a total of 36 records. After each set of nine records has been written, the program writes the associated subindex to the file. Then, one record indexed under the second subindex is read.

```
      PROGRAM MS4
      DIMENSION MASTER(5), SUBIX(10), REC(50)
C
      CALL OPENMS (2, MASTER, 5, 0)
      DO 10 MAJOR=1, 4
C
C     CLEAR SUBINDEX AREA
          DO 20 I=1, 10
          SUBIX(I)=0
 20       CONTINUE
C
C     MAKE SUBIX THE CURRENT INDEX
      CALL STINDX (2, SUBIX, 10)
C
C     GENERATE AND WRITE 9 RECORDS
      DO 30 MINOR=1, 9
           :
          CALL WRITMS (2, REC, 50, MINOR)
 30   CONTINUE
C
C     CHANGE CURRENT INDEX BACK TO MASTER INDEX
      CALL STINDX (2, MASTER, 5)
C
C     WRITE THE SUBINDEX TO THE FILE.
      CALL WRITMS (2, SUBIX, 10, MAJ, 0, 1)
 10   CONTINUE
C
C     READ RECORD 5 INDEXED UNDER THE 2ND SUBINDEX.
      CALL READMS (2, SUBIX, 10, 2)
      CALL STINDX (2, SUBIX, 10)
      CALL READMS (2, REC, 50, 5)
        :  (Manipulate the selected record as desired)
      END
```

Figure 14-10. Program MS4

Program SCLCALL

Program SCLCALL, shown in figure 14-11, illustrates the use of SCL interface calls to reference parameters specified on the execution call command.

```
      PROGRAM SCLCALL
C
C     DEFINE A STRING PARAMETER AND AN INTEGER VARIABLE PARAMETER.
C
C$    PARAM ('P1:STRING; P2:VAR OF INTEGER')
      CHARACTER KIND*8, SVAL*20, VREF*7, VARKND*7
      LOGICAL TSTPARM
      INTEGER VAL, RAD
C
C     TEST FOR PRESENCE OF P1 ON EXECUTION COMMAND.
C
      IF (TSTPARM('P1')) THEN
C
C     GET LENGTH AND VALUE OF P1.  ARGUMENTS setnum AND valnum ARE
C     NOT USED AND ARE SET TO 1.  ARGUMENT lhi IS NOT USED AND IS
C     SET TO 'LOW'.
C
          CALL GETCVAL ('P1', 1, 1, 'LOW', LEN, SVAL)
          PRINT 100, SVAL, LEN
 100      FORMAT (' PARAMETER NAME IS P1', /, ' VALUE IS ', A20,
     +/, ' LENGTH IS ', I2, /)
      ELSE
          PRINT*, ' PARAMETER P1 NOT SPECIFIED.'
      ENDIF
C
C     TEST FOR PRESENCE OF P2 ON EXECUTION COMMAND.
C
      IF (TSTPARM('P2')) THEN
C
C     GET NAME AND KIND OF VARIABLE REFERENCE.  ARGUMENTS setnum
C     AND valnum ARE NOT USED AND ARE SET TO 1.  ARGUMENT lhi IS NOT
C     USED AND IS SET TO 'LOW'.
C
          CALL GETVREF ('P2', 1, 1, 'LOW', VREF, VARKND, J, K, L)
C
C     GET VALUE AND BASE OF INTEGER VARIABLE.
C
          CALL REDIVAR (VREF, 1, VAL, RAD)
C
C     PRINT NAME, KIND, AND BASE OF INTEGER VARIABLE.
C
          PRINT 200, VREF, VARKND, RAD
 200      FORMAT (' VARIABLE NAME IS ', A7, /, ' KIND IS ', A7, /,
     +    ' BASE IS ', I2)
```

Figure 14-11. Program SCLCALL

(Continued)

(Continued)

```
C
C     DETERMINE BASE OF VALUE, THEN PRINT USING PROPER FORMAT.
C
          IF (RAD .EQ. 2 .OR. RAD .EQ. 8) PRINT 201, VAL
 201          FORMAT (' VALUE IS ', I20)
          IF (RAD .EQ. 10) PRINT 202, VAL
 202          FORMAT (' VALUE IS ', I10)
          IF (RAD .EQ. 16) PRINT 203, VAL
 203          FORMAT (' VALUE IS ', Z16)
      ELSE
          PRINT*, ' PARAMETER P2 NOT SPECIFIED.'
      ENDIF
      END
```

Figure 14-11. Program SCLCALL

The C$ PARAM directive defines a string parameter P1, and an integer variable parameter P2. The succeeding SCL calls test for the presence of the parameters on the execution command and, if the parameters are present, return information about the parameters.

Note that in the GETCVAL and GETVREF calls, the values 1, 1, and 'LOW' are supplied for the value number, value set number, and range position arguments, respectively. Even though value set, value list, and range attributes are not defined for P1 and P2, arguments corresponding to those attributes must be specified in the SCL calls.

Figure 14-12 shows examples of two execution commands for program SCLCALL. (The object code is assumed to be on file LGO.) In the first example, no parameters are specified. In the second example, a CREATE_VARIABLE command is entered to define an SCL integer variable named VAR, and the string parameter P1 and variable parameter P2 are specified on the LGO command. (User input is shown in blue.)

```
Example 1

    /lgo
     PARAMETER P1 NOT SPECIFIED.
     PARAMETER P2 NOT SPECIFIED.


Example 2

    /create_variable var kind=integer value=3A(16)
    /lgo p1='abcde' p2=var
    PARAMETER NAME IS P1
    VALUE IS abcde
    LENGTH IS  5

    VARIABLE NAME IS VAR
    KIND IS INTEGER
    BASE IS 16
    VALUE IS 000000000000003A
```

Figure 14-12. Sample Terminal Dialog for Program SCLCALL

Appendixes

Glossary A-1

Related Manuals B-1

ASCII Character Set and Collating Weight Tables C-1

Calling Other Language Subprograms D-1

Introduction to Debug E-1

The Programming Environment and the Professional Programming Environment F-1

Control Data Extensions to Standard FORTRAN G-1

Language Summary H-1

Differences Between FORTRAN Version 1 and FORTRAN Version 2 .... I-1

Glossary A

This section presents a list of definitions of terms used in this manual. It does not include terms defined in the ANSI standard for FORTRAN, X3.9-1978. Terms are listed in alphabetical order.

A

Absolute Value

A number that has been stripped of its sign.

ANSI Standard Language

The language defined by the American National Standards Institute. For FORTRAN, it is defined in American National Standard X3.9-1978.

Array Section

A group of elements of an array; an array section can contain one element, several elements, or all elements of an array.

ASCII

American National Standard Code for Information Interchange. It is a 7-bit code representing a prescribed set of characters. NOS/VE stores each 7-bit ASCII code right-justified in an 8-bit byte.

Attach

The process of retrieving a permanent file for use by a job. The process involves specifying the proper file identification and, if necessary, password.

B

Batch Mode

An execution mode in which a job is submitted and processed as a unit with no intervention from the user. Compare with Interactive Mode.

Beginning-of-Information (BOI)

The point at which data begins in a file.

Binary Object Program

An executable machine language program that is produced from a source program from the compiler.

Bit

A binary digit. It has the value 0 or 1. See Byte.

Blank Common Block

An unnamed common block. No data can be stored into a blank common block at load time. Contrast with Named Common Block.

Buffer Statement

One of the input/output statements BUFFER IN or BUFFER OUT.

Byte

A contiguous group of bits. A NOS/VE word has 8 bytes having 8 bits each. NOS/VE stores each ASCII character code in the rightmost 7 bits of a byte. The leftmost bit is used as a sign bit.

C

Call-By-Reference

A parameter passing style where the address of an actual parameter is passed to the corresponding formal parameter. Compare with Call-By-Value.

Call-By-Value

A parameter passing style where the value of an actual parameter is stored in a temporary memory location. The address of this temporary memory location, rather than the address of the actual parameter, is passed to the corresponding formal parameter. Compare with Call-By-Reference.

Called Program

A program that is the object of a CALL statement.

Calling Program

A program that executes a CALL statement.

Calling Sequence

A set of instructions used to transfer control to a subprogram.

Catalog

1. A directory of files and catalogs maintained by the system for a user. The catalog $LOCAL contains only temporary file entries.

2. The part of a path that identifies a particular catalog in a catalog hierarchy. The format is as follows:

   name.name. ... .name

   where each name is a catalog. See also Catalog Name and Path.

Catalog Name

The name of a catalog hierarchy (path). By convention, the name of the user's master catalog is the same as the user's user name.

Character

A letter, digit, or symbol represented by a code in a character set. Also, a unit of measure used to specify block length, record length, and so forth. Can be a nonprinting symbol or overpunch representation.

Close Operation

A set of terminating operations performed on a file when input and output operations are complete.

Close Request

A program request notifying the system that the program no longer intends to access file data through the specified instance of open. In response, the system flushes all modified data from memory to the file and ends the connection between the program and the file.

Collating Sequence

A set of values defining the collation weights of the 256 ASCII characters. The collation weights determine the sequence in which characters are ordered and their relative values when compared.

Collation Table

A data structure defining a collating sequence.

Collation Weight

The value assigned to a character that determines the position of that character when ordered using the collating sequence.

Common Block

An area of memory that can be declared in a COMMON statement by more than one program and used for storage of shared data. See Blank Common Block and Named Common Block.

Compilation Time

The time at which a source program is translated by the compiler to an object program that can be loaded and executed. Contrast with Execution Time.

Condition Code

Alphanumeric characters that uniquely identify a NOS/VE diagnostic. The condition code is returned as part of the status record when an abnormal status occurs.

Condition Name

A string value that corresponds to a specified condition code. Condition codes and names are used in status processing.

D

Default Data Type

The data type assumed by a variable in the absence of any type declarations for the variable. Variables whose names begin with one of the letters A through H or O through Z have a default type of real. Variables whose names begin with one of the letters I through N have a default type of integer. The default typing can be changed by using an IMPLICIT statement.

Default Value

The value used for the parameter value if no value is explicitly specified.

Diagnostic

An error message.

Direct Access Input/Output

A method of input/output in which records can be read or written in any order.

Display Code

A 64-character subset of the ASCII code, which consists of alphabetic letters, symbols, and numerals.

Domain

The domain of a function is the set containing all the arguments for which that function produces a result and not an error. For dyadic functions, the set is thought of as having argument pairs.

E

EBCDIC

The abbreviation for extended binary-coded decimal interchange code, an 8-bit code representing a coded character set.

End-of-File (EOF)

A particular kind of boundary on a sequential file, recognized by the END= specifier and the functions EOF and UNIT. Either of the following boundaries is recognized as end-of-file:

End-of-partition

End-of-information (EOI)

The ENDFILE statement writes an end-of-partition boundary.

End-of-Information (EOI)

The point at which data in a file ends.

End-of-Partition (EOP)

A special delimiter in a file with variable record type.

Entry Point

A location within a program unit that can be branched to from other program units. Each entry point has a unique name.

Equivalence Class

A group of variables or arrays where position relative to each other is defined as a result of an EQUIVALENCE statement.

Execution Time

The time at which a compiled program is executed. Also known as runtime. Contrast with compile time.

Extensible Common

A common block that has no upper bound other than the size of the memory segment it is contained in.

External File

A file residing on an external storage device. See File.

External Reference

A reference in one program unit to an entry point in another program unit.

External Storage Device

Disk or magnetic tape.

F

F Record Type

Fixed-length records, as defined by the ANSI standard.

Field

A subdivision of a record that consists of one or more contiguous characters.

File

A collection of information referenced by a name. Files read and written by FORTRAN programs can be classified according to their residence (external and internal files) or their method of access (sequential and direct access files). A file is identified by specifying a path and, optionally, a cycle reference (for permanent files) as follows:

path.cycle_reference

File Attribute

A characteristic of a file. Each file has a set of attributes that define the file structure and processing limitations.

File Cycle

A version of a file. All cycles of a file share the same file entry in a catalog. The file cycle is specified in a file reference by its number or by a special indicator, such as $NEXT.

File Position

The location in the file at which the next read or write operation begins. The file position designators are:

$ASIS	Leave the file in its current position.
$BOI	Position the file at the beginning-of-information.
$EOI	Position the file at the end-of-information.

File Reference

An SCL element that identifies a file and optionally the file position to be established prior to the file's use. The format of a file reference is:

:family.catalog.file.cycle.file_position

where file is a 1- to 31-character name.

where cycle is a numeric value from 1 to 999 that represents a version of the file.

where file_position is one of the following:

$ASIS
$BOI
$EOI

See also File and File Position.

Floating-Point Number

A method of internal binary representation for numbers written with a decimal point; corresponds to FORTRAN types REAL and DOUBLE PRECISION.

G

Generic Function Name

The name of an intrinsic function that accepts arguments of more than one data type. Except for data type conversion generic functions (and functions with boolean arguments), the type of the result is the same as the data type of the arguments.

Graphic Character

A character that can be printed or displayed.

I

Implicit Type

The type of a variable as declared in an IMPLICIT statement.

Indefinite Value

A value that results from a mathematical operation that cannot be resolved, such as a division where the dividend and divisor are both zero.

Infinite Value

A value that results from a computation whose result exceeds the largest value that can be represented in the computer. The representation of an infinite value in a computer word does not correspond to the representation of a number.

Inline Intrinsic Function

Functions that are expanded by the compiler rather than by a reference to the Math Library.

Instance of Open

A particular opening of a file as distinguished from all other openings of the same file. Closing the file ends the most recent instance of open.

Interactive Mode

A mode of execution where the user enters commands or data at the terminal during program execution.

Internal File

A character variable, array, or substring on which input/output operations are performed by formatted READ and WRITE statements. Internal files provide a method of transferring and converting data from one area of memory to another.

J

Job

A sequence of tasks executed for a user number.

Job Log

A chronological listing of all operations associated with a terminal or batch job from login to logout.

K

Keyed File Organization

A file organization that provides for record access by a key value. See the FORTRAN Keyed-File and Sort/Merge Interfaces manual.

Keyword

A word within a format that must be entered exactly as shown.

L

Library

See Source Library and Object library.

Load Time

The time at which an object program is loaded into memory and prepared for execution.

Local File

A file accessed via the $LOCAL catalog. See also File, Path, and Local Path.

Local Path

Identifies a local file as follows:

    $LOCAL.file_name

Login

The process used at a terminal to gain access to the system.

Logout

The process used to end a terminal session.

M

Mass Storage

Disk storage that allows random file access and permanent file storage.

Mass Storage File

A particular kind of randomly accessible file, accessed by the mass storage input/output routines.

Mass Storage Input/Output

A type of input/output used for random access to files; it involves the subroutines OPENMS, READMS, WRITMS, CLOSMS, and STINDX.

Math Library

A collection of mathematical routines on the system that can be called by the FORTRAN compiler.

Module

A unit of code. An object module is the unit of object code corresponding to a compilation unit. A load module is a unit of object code stored in an object library.

When using the Debug utility, module refers to a program unit.

N

Named Common Block

A common block that has a name. Data can be stored into a named common block at load time. The first program unit declaring a named common block determines the amount of memory allocated. Contrast with Blank Common Block.

Normalized Floating Point Number

A floating point number with the most significant bit of the fractional portion being nonzero.

NOS/VE

Acronym for Network Operating System/Virtual Environment, an operating system for the host computer.

O

Object Code

Executable code produced by the compiler. See Binary Object Program.

Object Library

A file containing one or more load, SCL procedures, program description, message, and/or load modules and a dictionary to each module.

Object Module

A compiler-generated unit containing object code and instructions for loading the object code. It is accepted as input by the system loader and the CREATE_OBJECT_LIBRARY utility.

Open

A set of preparatory operations performed on a file before input and output can take place.

Optimization

The manipulation of object code to reduce execution time. You can select the level of optimization performed by the compiler through the OPTIMIZATION_LEVEL parameter on the VECTOR_FORTRAN command.

P

Parameter

An item of information that is passed to or from a procedure or function.

Pass by Reference

A method of referencing a subprogram in which the addresses of the actual arguments are passed.

Pass by Value

A method of referencing a subprogram in which only the values of the actual arguments are passed.

Path

Identifies a file. A path may include the family name, user name, subcatalog name or names, and file name.

Permanent File

A file preserved by NOS/VE across job executions and system deadstarts. A permanent file has an entry in a permanent catalog. See File.

Procedure

A FORTRAN function subprogram, subroutine subprogram, or statement function.

Program-Library List

The list of object libraries searched for modules during program loading. A program-library list search is required to load a collation table module.

Program Unit

A sequence of FORTRAN statements terminated by an END statement. The FORTRAN program units are main programs, subroutines, functions, and block data subprograms.

Programming Environment

A full screen utility that allows you to create, modify, and execute programs on NOS/VE.

R

Random Access

The process of reading or writing a record in a file without having to read or write the preceding records; applies only to mass storage files. Contrast with Sequential Access.

Range

The range of a function is the set of all results produced by a function when it is applied to every argument in its domain.

Record

A unit of data that can be read or written by a single I/O request. Also, a set of related data processed as a unit when reading or writing a file.

Record Length

The length of a record measured in words for unformatted input/output and in characters for formatted input/output.

Reference Listing

A part of the listing produced by a FORTRAN compilation, which displays some or all of the entities used by the program, and provides other information such as attributes and location of those entities.

Relocatable

An object program that can reside in any part of memory. The actual starting address is established at load time.

Rewind

To position a file at its beginning-of-information.

Runtime

The time at which a compiled program is executed; also known as execution time.

S

Scalar

A scalar is a single value, such as a constant, variable, array element or substring.

SCL

See System Command Language.

Segment Access File

A file that is mapped to named common block. You access the file directly through the common block's variables and arrays using assignment statements.

Sequential Access

An access mode in which records are processed in the order (physical or logical) in which they occur on a storage device. Contrast with Random Access.

Sign

Indicates whether a number is positive or negative. A sign is one of the following characters:

+	Positive number
–	Negative number
space	Positive number

Source Code

Code written by the programmer in a language such as FORTRAN, and input to a compiler.

Source Library

A collection of text units on a file, generated and manipulated by the Source Code Utility (SCU).

Source Listing

A compiler-produced listing of the user's original source program.

Specific Function Name

The name of an intrinsic function that accepts arguments of a particular data type, and returns a result of a particular data type. Contrast with Generic Function Name.

Status Condition Code

The six-digit code that uniquely identifies a NOS/VE runtime diagnostic.

Status Variable

The variable in which the completion status of the command or procedure is returned.

String

A value that represents a sequence of characters.

Substring

A string variable consisting of zero or more consecutive character positions within a given string variable.

System Command Language (SCL)

The language that provides the interface to the features and capabilities of NOS/VE. All commands and statements are interpreted by SCL before being processed by the system.

T

Task

The instance of execution of a program.

Traceback

A list of subprogram names within a program, beginning with the currently executing subprogram, proceeding backward through the sequence of called subprograms, and ending with the main program.

U

U Record Type

Records for which the record structure is undefined.

Unit Identifier

An integer constant, or an integer variable with a value of either 0 to 999, an L format unit file name, or a segment access file name. In input/output statements, it indicates on which unit the operation is to be performed. It can be linked with the actual file name by an OPEN statement. If no OPEN statement is specified, a default file name is used.

User Name

A name that identifies a NOS/VE user and the location of the user's permanent files in the user's family.

User Path

Identifies a file or catalog via a user name and, optionally, a relative path as follows:

.user_name.relative_path

or

$USER.relative_path

Utility

A NOS/VE processor consisting of routines that perform a specific operation.

V

V Record Type

Variable-sized record; system default record type. Each V-type record has a record header. The header contains the record length and the length of the preceding record.

Vector

A one-dimensional set of numbers.

Vectorization

The manipulation of object code to reduce execution time that takes advantage of the vector-processing capabilities of the CYBER 180 model 990. You can select vectorization of your program through the VECTORIZATION_LEVEL option on the compilation command.

W

Word

Eight bytes.

Working Catalog

The working catalog is the catalog used if no other catalog is specified on a file reference. The $LOCAL catalog is the default working catalog. You can change the working catalog with the SET_WORKING_CATALOG command.

Working Storage Area

An area allocated by the task to hold data copied by get or put calls to a file.

Related Manuals B

Table B-1 lists all manuals that are referenced in this manual or that contain background information. A complete list of NOS/VE manuals is given in the NOS/VE System Usage manual. If your site has installed the online manuals, you can find an abstract for each NOS/VE manual in the online System Information manual. To access this manual, enter:

```
/help
```

Ordering Printed Manuals

Your can order Control Data manuals through Control Data sales offices or through:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

Accessing Online Manuals

To access an online manual, log in to NOS/VE and specify the online manual title (listed in table B-1) on the HELP or EXPLAIN command. To read online manuals written in TOPICS, your terminal must support full-screen mode. See the NOS/VE System Usage manual for information about terminal definitions that support the full screen interface. To read the VFORTRAN quick reference online manual, enter:

```
/help manual=vfortran
```

Table B-1. Related Manuals

Manual Title	Publication Number	Online Title
Background Manuals:		
NOS/VE Commands and Functions	60464018	SCL[1]
NOS/VE System Usage	60464014	
FORTRAN Manuals:		
FORTRAN Version 2 for NOS/VE Quick Reference		VFORTRAN
FORTRAN for NOS/VE Tutorial	60485912	FORTRAN_T
FORTRAN Version 1 for NOS/VE Language Definition Usage	60485913	
FORTRAN for NOS/VE Topics for FORTRAN Programmers Usage	60485916	
FORTRAN Version 1 for NOS/VE Quick Reference		FORTRAN
FORTRAN for NOS/VE Summary	60485919	
FORTRAN for NOS/VE Keyed File and Sort/Merge Interfaces	60485917	

(Continued)

Table B-1. Related Manuals *(Continued)*

Manual Title	Publication Number	Online Title
Additional References:		
Debug for NOS/VE Quick Reference		
Debug for NOS/VE Usage	60488213	
Math Library for NOS/VE Usage	60486513	
Migration From NOS to NOS/VE Tutorial/Usage	60489503	
NOS/VE Advanced File Management Usage	60486413	AFM
NOS/VE Diagnostic Messages	60464613	MESSAGES
NOS/VE Object Code Management Usage	60464413	OCM[1]
NOS/VE Online Examples		EXAMPLES[1]
NOS/VE Source Code Management Usage	60464313	SCM[1]
Professional Programming Environment for NOS/VE Quick Reference		PPE[1]
Programming Environment for NOS/VE Summary/Tutorial	60486819	
Programming Environment for NOS/VE Usage		ENVIRONMENT
Virtual State Volume II Hardware Reference	60458890	

1. These online manuals must be accessed in full-screen mode.

ASCII Character Set and Collating Weight Tables C

Tables C-1 through C-12 give the ASCII character set, the hexadecimal character code for each ASCII character, and the weight tables for the following collating sequences:

- ASCII: FORTRAN default collating sequence
- OSV$ASCII6_FOLDED and OSV$ASCII6_STRICT: NOS FORTRAN 5 default collating sequence.
- OSV$COBOL6_FOLDED and OSV$COBOL6_STRICT: NOS COBOL 5 default collating sequence.
- OSV$DISPLAY63_FOLDED and OSV$DISPLAY63_STRICT: NOS 63-character display code collating sequence.
- OSV$DISPLAY64_FOLDED and OSV$DISPLAY64_STRICT: NOS 64-character display code collating sequence.
- OSV$EBCDIC: Full EBCDIC collating sequence.
- OSV$EBCDIC6_FOLDED and OSV$EBCDIC6_STRICT: EBCDIC 6-bit subset collating sequence supported by NOS COBOL 5 and SORT 5.

The collation table variants FOLDED and STRICT indicate different mapping of the characters not in the 63 or 64 characters of the original NOS collating sequence. A strict mapping maps all characters not in the original 64 or 63-character set to the ordinal for the space character. A folded mapping maps some characters into ordinals of the original characters and the others into the ordinal value for the space character as shown in the listing of the collating sequence.

The following table shows the COLSEQ call parameter values and their corresponding weight table selection:

COLSEQ Call Parameter Value	Selected Collating Weight Table
ASCII	Standard ASCII
ASCII6	OSV$ASCII6_FOLDED
ASCII6S	OSV$ASCII6_STRICT
COBOL6	OSV$COBOL6_FOLDED
COBOL6S	OSV$COBOL6_STRICT
DISPLAY	OSV$DISPLAY64_FOLDED
DISPLAYS	OSV$DISPLAY64_STRICT
DISPLAY63	OSV$DISPLAY63_FOLDED
DISPLAY63S	OSV$DISPLAY63_STRICT
EBCDIC	OSV$EBCDIC
EBCDIC6	OSV$EBCDIC6_FOLDED
EBCDIC6S	OSV$EBCDIC6_STRICT
INSTALL	OSV$COBOL6_FOLDED

Table C-1. ASCII Character Set and Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
0	00	NULL	Null
1	01	SOH	Start of heading
2	02	STX	Start of text
3	03	ETX	End of text
4	04	EOT	End of transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal tabulation
10	0A	LF	Line feed
11	0B	VT	Vertical tabulation
12	0C	FF	Form feed
13	0D	CR	Carriage return
14	0E	SO	Shift out
15	0F	SI	Shift in
16	10	DLE	Data link escape
17	11	DC1	Device control 1
18	12	DC2	Device control 2
19	13	DC3	Device control 3
20	14	DC4	Device control 4
21	15	NAK	Negative acknowledge
22	16	SYN	Synchronous idle
23	17	ETB	End of transmission block
24	18	CAN	Cancel
25	19	EM	End of medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	SP	Space
33	21	!	Exclamation point
34	22	"	Quotation marks
35	23	#	Number sign
36	24	$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand
39	27	'	Apostrophe

(Continued)

Table C-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	28	(	Opening parenthesis
41	29	)	Closing parenthesis
42	2A	*	Asterisk
43	2B	+	Plus
44	2C	,	Comma
45	2D	-	Hyphen
46	2E	.	Period
47	2F	/	Slant
48	30	0	Zero
49	31	1	One
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less than
61	3D	=	Equal to
62	3E	>	Greater than
63	3F	?	Question mark
64	40	@	Commercial at
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O

(Continued)

Table C-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Opening bracket
92	5C	\	Reverse slant
93	5D	]	Closing bracket
94	5E	^	Circumflex
95	5F	_	Underline
96	60	`	Grave accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w

(Continued)

Table C-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Opening brace
124	7C	|	Vertical line
125	7D	}	Closing brace
126	7E	~	Tilde
127	7F	DEL	Delete

ASCII codes 80 through FF hexadecimal (not listed in this table) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-2. OSV$ASCII6_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	21	!	Exclamation point
02	22	"	Quotation marks
03	23	#	Number sign
04	24	$	Dollar sign
05	25	%	Percent sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	28	(	Opening parenthesis
09	29	)	Closing parenthesis
10	2A	*	Asterisk
11	2B	+	Plus
12	2C	,	Comma
13	2D	-	Hyphen
14	2E	.	Period
15	2F	/	Slant
16	30	0	Zero
17	31	1	One
18	32	2	Two
19	33	3	Three
20	34	4	Four
21	35	5	Five
22	36	6	Six
23	37	7	Seven
24	38	8	Eight
25	39	9	Nine
26	3A	:	Colon
27	3B	;	Semicolon
28	3C	<	Less than
29	3D	=	Equals
30	3E	>	Greater than
31	3F	?	Question mark
32	40,60	@,`	Commercial at, grave accent
33	41,61	A,a	Uppercase A, lowercase a
34	42,62	B,b	Uppercase B, lowercase b
35	43,63	C,c	Uppercase C, lowercase c
36	44,64	D,d	Uppercase D, lowercase d
37	45,65	E,e	Uppercase E, lowercase e
38	46,66	F,f	Uppercase F, lowercase f
39	47,67	G,g	Uppercase G, lowercase g

(Continued)

Table C-2. OSV$ASCII6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	48,68	H,h	Uppercase H, lowercase h
41	49,69	I,i	Uppercase I, lowercase i
42	4A,6Ai	J,j	Uppercase J, lowercase j
43	4B,6B	K,k	Uppercase K, lowercase k
44	4C,6C	L,l	Uppercase L, lowercase l
45	4D,6D	M,m	Uppercase M, lowercase m
46	4E,6E	N,n	Uppercase N, lowercase n
47	4F,6F	O,o	Uppercase O, lowercase o
48	50,70	P,p	Uppercase P, lowercase p
49	51,71	Q,q	Uppercase Q, lowercase q
50	52,72	R,r	Uppercase R, lowercase r
51	53,73	S,s	Uppercase S, lowercase s
52	54,74	T,t	Uppercase T, lowercase t
53	55,75	U,u	Uppercase U, lowercase u
54	56,76	V,v	Uppercase V, lowercase v
55	57,77	W,w	Uppercase W, lowercase w
56	58,78	X,x	Uppercase X, lowercase x
57	59,79	Y,y	Uppercase Y, lowercase y
58	5A,7A	Z,z	Uppercase Z, lowercase z
59	5B,7B	[,{	Opening bracket, opening brace
60	5C,7C	\\,|	Reverse slant, vertical line
61	5D,7D	],}	Closing bracket, closing brace
62	5E,7E	^,~	Circumflex, tilde
63	5F	_	Underline

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-3. OSV$ASCII6_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	21	!	Exclamation point
02	22	"	Quotation marks
03	23	#	Number sign
04	24	$	Dollar sign
05	25	%	Percent sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	28	(	Opening parenthesis
09	29	)	Closing parenthesis
10	2A	*	Asterisk
11	2B	+	Plus
12	2C	,	Comma
13	2D	-	Hyphen
14	2E	.	Period
15	2F	/	Slant
16	30	0	Zero
17	31	1	One
18	32	2	Two
19	33	3	Three
20	34	4	Four
21	35	5	Five
22	36	6	Six
23	37	7	Seven
24	38	8	Eight
25	39	9	Nine
26	3A	:	Colon
27	3B	;	Semicolon
28	3C	<	Less than
29	3D	=	Equals
30	3E	>	Greater than
31	3F	?	Question mark
32	40	@	Commercial at
33	41	A	Uppercase A
34	42	B	Uppercase B
35	43	C	Uppercase C
36	44	D	Uppercase D
37	45	E	Uppercase E
38	46	F	Uppercase F
39	47	G	Uppercase G

(Continued)

Table C-3. OSV$ASCII6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	48	H	Uppercase H
41	49	I	Uppercase I
42	4A	J	Uppercase J
43	4B	K	Uppercase K
44	4C	L	Uppercase L
45	4D	M	Uppercase M
46	4E	N	Uppercase N
47	4F	O	Uppercase O
48	50	P	Uppercase P
49	51	Q	Uppercase Q
50	52	R	Uppercase R
51	53	S	Uppercase S
52	54	T	Uppercase T
53	55	U	Uppercase U
54	56	V	Uppercase V
55	57	W	Uppercase W
56	58	X	Uppercase X
57	59	Y	Uppercase Y
58	5A	Z	Uppercase Z
59	5B	[	Opening bracket
60	5C	\	Reverse slant
61	5D	]	Closing bracket
62	5E	^	Circumflex
63	5F	_	Underline

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-4. OSV$COBOL6_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	40,60	@,`	Commercial at, grave accent
02	25	%	Percent sign
03	5B,7B	[,{	Opening bracket, opening brace
04	5F	_	Underline
05	23	#	Number sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	3F	?	Question mark
09	3E	>	Greater than
10	5C,7C	\\,|	Reverse slant, vertical line
11	5E,7E	^,~	Circumflex, tilde
12	2E	.	Period
13	29	)	Closing parenthesis
14	3B	;	Semicolon
15	2B	+	Plus
16	24	$	Dollar sign
17	2A	*	Asterisk
18	2D	-	Hyphen
19	2F	/	Slant
20	2C	,	Comma
21	28	(	Opening parenthesis
22	3D	=	Equals
23	22	"	Quotation marks
24	3C	<	Less than
25	41,61	A,a	Uppercase A, lowercase a
26	42,62	B,b	Uppercase B, lowercase b
27	43,63	C,c	Uppercase C, lowercase c
28	44,64	D,d	Uppercase D, lowercase d
29	45,65	E,e	Uppercase E, lowercase e
30	46,66	F,f	Uppercase F, lowercase f
31	47,67	G,g	Uppercase G, lowercase g
32	48,68	H,h	Uppercase H, lowercase h
33	49,69	I,i	Uppercase I, lowercase i
34	21	!	Exclamation point
35	4A,6A	J,j	Uppercase J, lowercase j
36	4B,6B	K,k	Uppercase K, lowercase k
37	4C,6C	L,l	Uppercase L, lowercase l
38	4D,6D	M,m	Uppercase M, lowercase m
39	4E,6E	N,n	Uppercase N, lowercase n

(Continued)

Table C-4. OSV$COBOL6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4F,6F	O,o	Uppercase O, lowercase o
41	50,70	P,p	Uppercase P, lowercase p
42	51,71	Q,q	Uppercase Q, lowercase q
43	52,72	R,r	Uppercase R, lowercase r
44	5D,7D	],}	Closing bracket, closing brace
45	53,73	S,s	Uppercase S, lowercase s
46	54,74	T,t	Uppercase T, lowercase t
47	55,75	U,u	Uppercase U, lowercase u
48	56,76	V,v	Uppercase V, lowercase v
49	57,77	W,w	Uppercase W, lowercase w
50	58,78	X,x	Uppercase X, lowercase x
51	59,79	Y,y	Uppercase Y, lowercase y
52	5A,7A	Z,z	Uppercase Z, lowercase z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-5. OSV$COBOL6_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	40	@	Commercial at
02	25	%	Percent sign
03	5B	[	Opening bracket
04	5F	_	Underline
05	23	#	Number sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	3F	?	Question mark
09	3E	>	Greater than
10	5C	\	Reverse slant
11	5E	^	Circumflex
12	2E	.	Period
13	29	)	Closing parenthesis
14	3B	;	Semicolon
15	2B	+	Plus
16	24	$	Dollar sign
17	2A	*	Asterisk
18	2D	-	Hyphen
19	2F	/	Slant
20	2C	,	Comma
21	28	(	Opening parenthesis
22	3D	=	Equals
23	22	"	Quotation marks
24	3C	<	Less than
25	41	A	Uppercase A
26	42	B	Uppercase B
27	43	C	Uppercase C
28	44	D	Uppercase D
29	45	E	Uppercase E
30	46	F	Uppercase F
31	47	G	Uppercase G
32	48	H	Uppercase H
33	49	I	Uppercase I
34	21	!	Exclamation point
35	4A	J	Uppercase J
36	4B	K	Uppercase K
37	4C	L	Uppercase L
38	4D	M	Uppercase M
39	4E	N	Uppercase N

(Continued)

Table C-5. OSV$COBOL6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4F	O	Uppercase O
41	50	P	Uppercase P
42	51	Q	Uppercase Q
43	52	R	Uppercase R
44	5D	]	Closing bracket
45	53	S	Uppercase S
46	54	T	Uppercase T
47	55	U	Uppercase U
48	56	V	Uppercase V
49	57	W	Uppercase W
50	58	X	Uppercase X
51	59	Y	Uppercase Y
52	5A	Z	Uppercase Z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-6. OSV$DISPLAY63_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
01	41,61	A,a	Uppercase A, lowercase a
02	42,62	B,b	Uppercase B, lowercase b
03	43,63	C,c	Uppercase C, lowercase c
04	44,64	D,d	Uppercase D, lowercase d
05	45,65	E,e	Uppercase E, lowercase e
06	46,66	F,f	Uppercase F, lowercase f
07	47,67	G,g	Uppercase G, lowercase g
08	48,68	H,h	Uppercase H, lowercase h
09	49,69	I,i	Uppercase I, lowercase i
10	4A,6A	J,j	Uppercase J, lowercase j
11	4B,6B	K,k	Uppercase K, lowercase k
12	4C,6C	L,l	Uppercase L, lowercase l
13	4D,6D	M,m	Uppercase M, lowercase m
14	4E,6E	N,n	Uppercase N, lowercase n
15	4F,6F	O,o	Uppercase O, lowercase o
16	50,70	P,p	Uppercase P, lowercase p
17	51,71	Q,q	Uppercase Q, lowercase q
18	52,72	R,r	Uppercase R, lowercase r
19	53,73	S,s	Uppercase S, lowercase s
20	54,74	T,t	Uppercase T, lowercase t
21	55,75	U,u	Uppercase U, lowercase u
22	56,76	V,v	Uppercase V, lowercase v
23	57,77	W,w	Uppercase W, lowercase w
24	58,78	X,x	Uppercase X, lowercase x
25	59,79	Y,y	Uppercase Y, lowercase y
26	5A,7A	Z,z	Uppercase Z, lowercase z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk
40	2F	/	Slant

(Continued)

Table C-6. OSV$DISPLAY63_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B,7B	[,{	Opening bracket, opening brace
50	5D,7D	],}	Closing bracket, closing brace
51	3A	:	Colon
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40,60	@,`	Commercial at, grave accent
61	5C,7C	\,|	Reverse slant, vertical line
62	5E,7E	^,~	Circumflex, tilde
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F, 25, and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-7. OSV$DISPLAY63_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
01	41	A	Uppercase A
02	42	B	Uppercase B
03	43	C	Uppercase C
04	44	D	Uppercase D
05	45	E	Uppercase E
06	46	F	Uppercase F
07	47	G	Uppercase G
08	48	H	Uppercase H
09	49	I	Uppercase I
10	4A	J	Uppercase J
11	4B	K	Uppercase K
12	4C	L	Uppercase L
13	4D	M	Uppercase M
14	4E	N	Uppercase N
15	4F	O	Uppercase O
16	50	P	Uppercase P
17	51	Q	Uppercase Q
18	52	R	Uppercase R
19	53	S	Uppercase S
20	54	T	Uppercase T
21	55	U	Uppercase U
22	56	V	Uppercase V
23	57	W	Uppercase W
24	58	X	Uppercase X
25	59	Y	Uppercase Y
26	5A	Z	Uppercase Z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk
40	2F	/	Slant

(Continued)

Table C-7. OSV$DISPLAY63_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B	[	Opening bracket
50	5D	]	Closing bracket
51	3A	:	Colon
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40	@	Commercial at
61	5C	\	Reverse slant
62	5E	^	Circumflex
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F, 25, and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-8. OSV$DISPLAY64_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	3A	:	Colon
01	41,61	A,a	Uppercase A, lowercase a
02	42,62	B,b	Uppercase B, lowercase b
03	43,63	C,c	Uppercase C, lowercase c
04	44,64	D,d	Uppercase D, lowercase d
05	45,65	E,e	Uppercase E, lowercase e
06	46,66	F,f	Uppercase F, lowercase f
07	47,67	G,g	Uppercase G, lowercase g
08	48,68	H,h	Uppercase H, lowercase h
09	49,69	I,i	Uppercase I, lowercase i
10	4A,6A	J,j	Uppercase J, lowercase j
11	4B,6B	K,k	Uppercase K, lowercase k
12	4C,6C	L,l	Uppercase L, lowercase l
13	4D,6D	M,m	Uppercase M, lowercase m
14	4E,6E	N,n	Uppercase N, lowercase n
15	4F,6F	O,o	Uppercase O, lowercase o
16	50,70	P,p	Uppercase P, lowercase p
17	51,71	Q,q	Uppercase Q, lowercase q
18	52,72	R,r	Uppercase R, lowercase r
19	53,73	S,s	Uppercase S, lowercase s
20	54,74	T,t	Uppercase T, lowercase t
21	55,75	U,u	Uppercase U, lowercase u
22	56,76	V,v	Uppercase V, lowercase v
23	57,77	W,w	Uppercase W, lowercase w
24	58,78	X,x	Uppercase X, lowercase x
25	59,79	Y,y	Uppercase Y, lowercase y
26	5A,7A	Z,z	Uppercase Z, lowercase z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk

(Continued)

Table C-8. OSV$DISPLAY64_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B,7B	[,{	Opening bracket, opening brace
50	5D,7D	],}	Closing bracket, closing brace
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40,60	@,`	Commercial at, grave accent
61	5C,7C	\,|	Reverse slant, vertical line
62	5E,7E	^,~	Circumflex, tilde
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-9. OSV$DISPLAY64_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	3A	:	Colon
01	41	A	Uppercase A
02	42	B	Uppercase B
03	43	C	Uppercase C
04	44	D	Uppercase D
05	45	E	Uppercase E
06	46	F	Uppercase F
07	47	G	Uppercase G
08	48	H	Uppercase H
09	49	I	Uppercase I
10	4A	J	Uppercase J
11	4B	K	Uppercase K
12	4C	L	Uppercase L
13	4D	M	Uppercase M
14	4E	N	Uppercase N
15	4F	O	Uppercase O
16	50	P	Uppercase P
17	51	Q	Uppercase Q
18	52	R	Uppercase R
19	53	S	Uppercase S
20	54	T	Uppercase T
21	55	U	Uppercase U
22	56	V	Uppercase V
23	57	W	Uppercase W
24	58	X	Uppercase X
25	59	Y	Uppercase Y
26	5A	Z	Uppercase Z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk

(Continued)

Table C-9. OSV$DISPLAY64_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B	[	Opening bracket
50	5D	]	Closing bracket
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40	@	Commercial at
61	5C	\	Reverse slant
62	5E	^	Circumflex
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-10. OSV$EBCDIC Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
000	00	NUL	Null
001	01	SOH	Start of heading
002	02	STX	Start of text
003	03	ETX	End of text
004	9C	---	Unassigned
005	09	HT	Horizontal tabulation
006	86	---	Unassigned
007	7F	DEL	Delete
008	97	---	Unassigned
009	8D	---	Unassigned
010	8E	---	Unassigned
011	0B	VT	Vertical tabulation
012	0C	FF	Form feed
013	0D	CR	Carriage return
014	0E	SO	Shift out
015	0F	SI	Shift in
016	10	DLE	Data link escape
017	11	DC1	Device control 1
018	12	DC2	Device control 2
019	13	DC3	Device control 3
020	9D	---	Unassigned
021	85	---	Unassigned
022	08	BS	Backspace
023	87	---	Unassigned
024	18	CAN	Cancel
025	19	EM	End of medium
026	92	---	Unassigned
027	8F	---	Unassigned
028	1C	FS	File separator
029	1D	GS	Group separator
030	1E	RS	Record separator
031	1F	US	Unit separator
032	80	---	Unassigned
033	81	---	Unassigned
034	82	---	Unassigned
035	83	---	Unassigned
036	84	---	Unassigned
037	0A	LF	Line feed
038	17	ETB	End of transmission block
039	1B	ESC	Escape

(Continued)

Table C-9. OSV$DISPLAY64_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period
48	23	#	Number sign
49	5B	[	Opening bracket
50	5D	]	Closing bracket
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40	@	Commercial at
61	5C	\	Reverse slant
62	5E	^	Circumflex
63	3B	;	Semicolon

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-10. OSV$EBCDIC Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
000	00	NUL	Null
001	01	SOH	Start of heading
002	02	STX	Start of text
003	03	ETX	End of text
004	9C	---	Unassigned
005	09	HT	Horizontal tabulation
006	86	---	Unassigned
007	7F	DEL	Delete
008	97	---	Unassigned
009	8D	---	Unassigned
010	8E	---	Unassigned
011	0B	VT	Vertical tabulation
012	0C	FF	Form feed
013	0D	CR	Carriage return
014	0E	SO	Shift out
015	0F	SI	Shift in
016	10	DLE	Data link escape
017	11	DC1	Device control 1
018	12	DC2	Device control 2
019	13	DC3	Device control 3
020	9D	---	Unassigned
021	85	---	Unassigned
022	08	BS	Backspace
023	87	---	Unassigned
024	18	CAN	Cancel
025	19	EM	End of medium
026	92	---	Unassigned
027	8F	---	Unassigned
028	1C	FS	File separator
029	1D	GS	Group separator
030	1E	RS	Record separator
031	1F	US	Unit separator
032	80	---	Unassigned
033	81	---	Unassigned
034	82	---	Unassigned
035	83	---	Unassigned
036	84	---	Unassigned
037	0A	LF	Line feed
038	17	ETB	End of transmission block
039	1B	ESC	Escape

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
040	88	---	Unassigned
041	89	---	Unassigned
042	8A	---	Unassigned
043	8B	---	Unassigned
044	8C	---	Unassigned
045	05	ENQ	Enquiry
046	06	ACK	Acknowledge
047	07	BEL	Bell
048	90	---	Unassigned
049	91	---	Unassigned
050	16	SYN	Synchronous idle
051	93	---	Unassigned
052	94	---	Unassigned
053	95	---	Unassigned
054	96	---	Unassigned
055	04	EOT	End of transmission
056	98	---	Unassigned
057	99	---	Unassigned
058	9A	---	Unassigned
059	9B	---	Unassigned
060	14	DC4	Device control 4
061	15	NAK	Negative acknowledge
062	9E	---	Unassigned
063	1A	SUB	Substitute
064	20	SP	Space
065	A0	---	Unassigned
066	A1	---	Unassigned
067	A2	---	Unassigned
068	A3	---	Unassigned
069	A4	---	Unassigned
070	A5	---	Unassigned
071	A6	---	Unassigned
072	A7	---	Unassigned
073	A8	---	Unassigned
074	5B	[	Opening bracket
075	2E	.	Period
076	3C	<	Less than
077	28	(	Opening parenthesis
078	2B	+	Plus
079	21	!	Exclamation point

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
080	26	&	Ampersand
081	A9	---	Unassigned
082	AA	---	Unassigned
083	AB	---	Unassigned
084	AC	---	Unassigned
085	AD	---	Unassigned
086	AE	---	Unassigned
087	AF	---	Unassigned
088	B0	---	Unassigned
089	B1	---	Unassigned
090	5D	]	Closing bracket
091	24	$	Dollar sign
092	2A	*	Asterisk
093	29	)	Closing parenthesis
094	3B	;	Semicolon
095	5E	^	Circumflex
096	2D	-	Hyphen
097	2F	/	Slant
098	B2	---	Unassigned
099	B3	---	Unassigned
100	B4	---	Unassigned
101	B5	---	Unassigned
102	B6	---	Unassigned
103	B7	---	Unassigned
104	B8	---	Unassigned
105	B9	---	Unassigned
106	7C	|	Vertical line
107	2C	,	Comma
108	25	%	Percent sign
109	5F	_	Underline
110	3E	>	Greater than
111	3F	?	Question mark
112	BA	---	Unassigned
113	BB	---	Unassigned
114	BC	---	Unassigned
115	BD	---	Unassigned
116	BE	---	Unassigned
117	BF	---	Unassigned
118	C0	---	Unassigned
119	C1	---	Unassigned

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
120	C2	---	Unassigned
121	60	`	Grave accent
122	3A	:	Colon
123	23	#	Number sign
124	40	@	Commercial at
125	27	'	Apostrophe
126	3D	=	Equals
127	22	"	Quotation marks
128	C3	---	Unassigned
129	61	a	Lowercase a
130	62	b	Lowercase b
131	63	c	Lowercase c
132	64	d	Lowercase d
133	65	e	Lowercase e
134	66	f	Lowercase f
135	67	g	Lowercase g
136	68	h	Lowercase h
137	69	i	Lowercase i
138	C4	---	Unassigned
139	C5	---	Unassigned
140	C6	---	Unassigned
141	C7	---	Unassigned
142	C8	---	Unassigned
143	C9	---	Unassigned
144	CA	---	Unassigned
145	6A	j	Lowercase j
146	6B	k	Lowercase k
147	6C	l	Lowercase l
148	6D	m	Lowercase m
149	6E	n	Lowercase n
150	6F	o	Lowercase o
151	70	p	Lowercase p
152	71	q	Lowercase q
153	72	r	Lowercase r
154	CB	---	Unassigned
155	CC	---	Unassigned
156	CD	---	Unassigned
157	CE	---	Unassigned
158	CF	---	Unassigned
159	D0	---	Unassigned

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
160	D1	---	Unassigned
161	7E	---	Unassigned
162	73	s	Lowercase s
163	74	t	Lowercase t
164	75	u	Lowercase u
165	76	v	Lowercase v
166	77	w	Lowercase w
167	78	x	Lowercase x
168	79	y	Lowercase y
169	7A	z	Lowercase z
170	D2	---	Unassigned
171	D3	---	Unassigned
172	D4	---	Unassigned
173	D5	---	Unassigned
174	D6	---	Unassigned
175	D7	---	Unassigned
176	D8	---	Unassigned
177	D9	---	Unassigned
178	DA	---	Unassigned
179	DB	---	Unassigned
180	DC	---	Unassigned
181	DD	---	Unassigned
182	DE	---	Unassigned
183	DF	---	Unassigned
184	E0	---	Unassigned
185	E1	---	Unassigned
186	E2	---	Unassigned
187	E3	---	Unassigned
188	E4	---	Unassigned
189	E5	---	Unassigned
190	E6	---	Unassigned
191	E7	---	Unassigned
192	7B	{	Opening brace
193	41	A	Uppercase A
194	42	B	Uppercase B
195	43	C	Uppercase C
196	44	D	Uppercase D
197	45	E	Uppercase E
198	46	F	Uppercase F
199	47	G	Uppercase G

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
200	48	H	Uppercase H
201	49	I	Uppercase I
202	E8	---	Unassigned
203	E9	---	Unassigned
204	EA	---	Unassigned
205	EB	---	Unassigned
206	EC	---	Unassigned
207	ED	---	Unassigned
208	7D	}	Closing brace
209	4A	J	Uppercase J
210	4B	K	Uppercase K
211	4C	L	Uppercase L
212	4D	M	Uppercase M
213	4E	N	Uppercase N
214	4F	O	Uppercase O
215	50	P	Uppercase P
216	51	Q	Uppercase Q
217	52	R	Uppercase R
218	EE	---	Unassigned
219	EF	---	Unassigned
220	F0	---	Unassigned
221	F1	---	Unassigned
222	F2	---	Unassigned
223	F3	---	Unassigned
224	5C	\	Reverse slant
225	9F	---	Unassigned
226	53	S	Uppercase S
227	54	T	Uppercase T
228	55	U	Uppercase U
229	56	V	Uppercase V
230	57	W	Uppercase W
231	58	X	Uppercase X
232	59	Y	Uppercase Y
233	5A	Z	Uppercase Z
234	F4	---	Unassigned
235	F5	---	Unassigned
236	F6	---	Unassigned
237	F7	---	Unassigned
238	F8	---	Unassigned
239	F9	---	Unassigned

(Continued)

Table C-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
240	30	0	Zero
241	31	1	One
242	32	2	Two
243	33	3	Three
244	34	4	Four
245	35	5	Five
246	36	6	Six
247	37	7	Seven
248	38	8	Eight
249	39	9	Nine
250	FA	---	Unassigned
251	FB	---	Unassigned
252	FC	---	Unassigned
253	FD	---	Unassigned
254	FE	---	Unassigned
255	FF	---	Unassigned

Table C-11. OSV$EBCDIC6_FOLDED Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	2E	.	Period
02	3C	<	Less than
03	28	(	Opening parenthesis
04	2B	+	Plus
05	21	!	Exclamation point
06	26	&	Ampersand
07	24	$	Dollar sign
08	2A	*	Asterisk
09	29	)	Closing parenthesis
10	3B	;	Semicolon
11	5E,7E	^,~	Circumflex, tilde
12	2D	-	Hyphen
13	2F	/	Slant
14	2C	,	Comma
15	25	%	Percent sign
16	5F	_	Underline
17	3E	>	Greater than
18	3F	?	Question mark
19	3A	:	Colon
20	23	#	Number sign
21	40,60	@,`	Commercial at, grave accent
22	27	'	Apostrophe
23	3D	=	Equals
24	22	"	Quotation marks
25	5B,7B	[,{	Opening bracket, opening brace
26	41,61	A,a	Uppercase A, lowercase a
27	42,62	B,b	Uppercase B, lowercase b
28	43,63	C,c	Uppercase C, lowercase c
29	44,64	D,d	Uppercase D, lowercase d
30	45,65	E,e	Uppercase E, lowercase e
31	46,66	F,f	Uppercase F, lowercase f
32	47,67	G,g	Uppercase G, lowercase g
33	48,68	H,h	Uppercase H, lowercase h
34	49,69	I,i	Uppercase I, lowercase i
35	5D,7D	],}	Closing bracket, closing brace
36	4A,6A	J,j	Uppercase J, lowercase j
37	4B,6B	K,k	Uppercase K, lowercase k
38	4C,6C	L,l	Uppercase L, lowercase l
39	4D,6D	M,m	Uppercase M, lowercase m

(Continued)

Table C-11. OSV$EBCDIC6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4E,6E	N,n	Uppercase N, lowercase n
41	4F,6F	O,o	Uppercase O, lowercase o
42	50,70	P,p	Uppercase P, lowercase p
43	51,71	Q,q	Uppercase Q, lowercase q
44	52,72	R,r	Uppercase R, lowercase r
45	5C,7C	\\,|	Reverse slant, vertical line
46	53,73	S,s	Uppercase S, lowercase s
47	54,74	T,t	Uppercase T, lowercase t
48	55,75	U,u	Uppercase U, lowercase u
49	56,76	V,v	Uppercase V, lowercase v
50	57,77	W,w	Uppercase W, lowercase w
51	58,78	X,x	Uppercase X, lowercase x
52	59,79	Y,y	Uppercase Y, lowercase y
53	5A,7A	Z,z	Uppercase Z, lowercase z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table C-12. OSV$EBCDIC6_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	2E	.	Period
02	3C	<	Less than
03	28	(	Opening parenthesis
04	2B	+	Plus
05	21	!	Exclamation point
06	26	&	Ampersand
07	24	$	Dollar sign
08	2A	*	Asterisk
09	29	)	Closing parenthesis
10	3B	;	Semicolon
11	5E	^	Circumflex
12	2D	-	Hyphen
13	2F	/	Slant
14	2C	,	Comma
15	25	%	Percent sign
16	5F	_	Underline
17	3E	>	Greater than
18	3F	?	Question mark
19	3A	:	Colon
20	23	#	Number sign
21	40	@	Commercial at
22	27	'	Apostrophe
23	3D	=	Equals
24	22	"	Quotation marks
25	5B	[	Opening bracket
26	41	A	Uppercase A
27	42	B	Uppercase B
28	43	C	Uppercase C
29	44	D	Uppercase D
30	45	E	Uppercase E
31	46	F	Uppercase F
32	47	G	Uppercase G
33	48	H	Uppercase H
34	49	I	Uppercase I
35	5D	]	Closing bracket
36	4A	J	Uppercase J
37	4B	K	Uppercase K
38	4C	L	Uppercase L
39	4D	M	Uppercase M

(Continued)

Table C-12. OSV$EBCDIC6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	4E	N	Uppercase N
41	4F	O	Uppercase O
42	50	P	Uppercase P
43	51	Q	Uppercase Q
44	52	R	Uppercase R
45	5C	\	Reverse slant
46	53	S	Uppercase S
47	54	T	Uppercase T
48	55	U	Uppercase U
49	56	V	Uppercase V
50	57	W	Uppercase W
51	58	X	Uppercase X
52	59	Y	Uppercase Y
53	5A	Z	Uppercase Z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Calling Other Language Subprograms D

FORTRAN allows you to call subprograms written in languages other than FORTRAN Version 2. These languages include CYBIL, COBOL and C. All languages under NOS/VE generate a standard subprogram calling sequence. Therefore, the only restrictions are to ensure that the subprogram names and dummy arguments (if any) conform to FORTRAN requirements. The rules described in chapter 8, Program Units, for calling FORTRAN subprograms also apply to calling subprograms written in other languages.

You should be careful when accessing the same file both in a FORTRAN program and in a subprogram written in another language, because of the language dependency of certain operations. The following operations always produce the expected results when performed on shared files:

- All terminal input and output.
- All usages where output is the only operation being performed on the shared file.
- All usages where the file is written by one or more languages and then read by another language, provided that you close and then reopen the file before reading it.

Other usages can cause unexpected results and should be avoided.

Calling CYBIL Procedures

You can call a CYBIL procedure from a FORTRAN program using a standard CALL or function reference statement. Values can be passed between the CYBIL procedure and FORTRAN program through the argument list and function return value. If the CYBIL procedure is a valid FORTRAN name (1 through 31 letters, digits, or underscores, beginning with a letter), the procedure must be declared with the XDCL attribute in the CYBIL routine. If the CYBIL procedure name is not a valid FORTRAN name, you must use the C$ EXTERNAL statement to rename the procedure. The C$ EXTERNAL statement is described in chapter 11. The dummy (formal) parameters must be declared in VAR of the CYBIL routine and must correspond in number and data type to the FORTRAN actual arguments.

The arguments passed to the CYBIL subprogram can be variables, constants, symbolic constants, arrays, or expressions with operators.

The correspondence of data types between FORTRAN and CYBIL is as follows:

FORTRAN	CYBIL
INTEGER	INTEGER
LOGICAL	-
CHARACTER	STRING(*) or CHARACTER
COMPLEX	-
BOOLEAN	-
REAL	REAL
DOUBLE PRECISION	-
CHARACTER*264	OST$STATUS

The OST$STATUS variable of system-resident CYBIL procedures can be processed as a FORTRAN character variable by using the NOS/VE status subprograms described in chapter 10, NOS/VE and Utility Subprograms. The dummy arguments can be subranges of any of the above types.

NOTE

Arithmetic values returned to a FORTRAN procedure must be word aligned, that is, begin on a multiple of 8 bytes. Byte, 2-byte, and 4-byte integer values are not permitted.

The storage sequence for CYBIL arrays having dimension greater than one differs from that of FORTRAN. In FORTRAN, arrays are stored in columnwise order, whereas in CYBIL, similar structures are stored in rowwise order. Thus, for example, the elements of a FORTRAN array dimensioned (3,2) and a CYBIL array dimensioned [1..3, 1..2] are as follows:

(1,1)	(2,1)	(3,1)	(1,2)	(2,2)	(3,2)	FORTRAN array
[1][1]	[1][2]	[2][1]	[2][2]	[3][1]	[3][2]	CYBIL array

The following example shows a FORTRAN main program that calls a CYBIL procedure named CT. The main program reads a character string and a single character from the terminal and passes them to the CYBIL procedure. The CYBIL procedure counts the number of occurrences of the character within the string and returns the result to the main program. The main program prints the result and terminates.

```
      PROGRAM STRING
      CHARACTER STR*60, C

C     Request and read input values.

      PRINT *, ' Type input string'
      READ (*,*) STR
      PRINT *, ' Type character to be counted'
      READ (*,*) C

C     Call CYBIL procedure.

      CALL CT (C, STR, N)

C     Print results.

      PRINT 99, STR, C, N
 99   FORMAT (' Input string= ', A60, /' Character= ', A1,
     +      /' Count= ', I2)
      END
```

CYBIL Procedure:

```
MODULE char_count;
  PROCEDURE [XDCL] ct (VAR c: char;
                       VAR s: string(*);
                       VAR i: integer);
    VAR j: integer;
    i := 0;
    FOR j := 1 TO STRLENGTH(s) DO;
      IF s(j) = c THEN;
        i := i + 1;
      IFEND;
    FOREND;
  PROCEND ct;
MODEND;
```

Example of terminal dialog:

`/fortran input=cytest binary_object=ftnbin`	Compile FORTRAN main program.
`/cybil input=cyproc binary_object=cybin`	Compile CYBIL procedure.
`/execute_task (ftnbin,cybin)`	Execute program.
` Type input string`	Prompt for input.
`? 'Hi there, somebody'`	User input.
` Type character to be counted`	Prompt for input.
`?  'e'`	User input.

`Input string= Hi there, somebody`
`Character= e`
`Count= 3`
	Program output.

Calling COBOL Subprograms

You can call COBOL subprograms from a FORTRAN program using a standard CALL statement. The name of the COBOL subprogram must be a valid FORTRAN name (1 through 31 letters, digits, or underscores, beginning with a letter). You can pass values between the calling program and the called subprogram through an argument list or through common storage.

If you use an argument list, the actual arguments in the CALL statement are associated with dummy arguments specified in the USING clause of the PROCEDURE DIVISION statement. The actual arguments must correspond in number and data type to the dummy arguments. The rules for argument association described in chapter 8, Program Units, apply to COBOL subprograms as well.

The correspondence of data types between FORTRAN and COBOL is as follows:

FORTRAN	COBOL
REAL	COMP-1
DOUBLE PRECISION	COMP-2
INTEGER (signed)	S9(18) COMP SYNC LEFT
CHARACTER	X(n) or 9(n) or A(n)
COMPLEX	-
BOOLEAN	-
LOGICAL	-

Communication through common storage is achieved through the named common block CCOMMON. In the COBOL subprogram, all data assigned to the COMMON_STORAGE section is automatically placed in CCOMMON. You can share the common storage area by declaring CCOMMON in the FORTRAN calling program.

The following example shows how a FORTRAN program can call a simple COBOL subprogram. The FORTRAN program COBTST calls the COBOL subprogram COBSUB, and passes values through the argument list and through CCOMMON. An integer value and a real value are passed to CBSUB through CCOMMON, and a character string STR1 is passed through the argument list. CBSUB displays the two numbers and the string, and returns a string to COBTST through the argument list.

Program COBTST:

```
PROGRAM COBTST
CHARACTER STR1*21, STR2*9
COMMON /CCOMMON/I, X
STR1 = 'Here are some numbers'
I = 4
X = 2.414
```

Items in CCOMMON share storage with items in the COMMON-STORAGE section.

C Call COBOL subprogram:

```
CALL CBSUB (STR1, STR2)
PRINT *, STR2
END
```

Actual arguments are associated with items declared in LINKAGE section.

Subprogram CBSUB:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBSUB.

DATA DIVISION.
COMMON-STORAGE SECTION.
01 INTEGER-1   PIC 9(18) COMP SYNC LEFT.
01 REAL-1      COMP-1.
```

Define items to be in common.

```
LINKAGE SECTION.
77 STRING-1 PIC X(21).
77 STRING-2 PIC X(9).
```

Define items to be passed as arguments.

```
PROCEDURE DIVISION USING STRING-1, STRING-2.
PASSING-DATA.
    DISPLAY STRING-1.
    DISPLAY "First number = " INTEGER-1.
    DISPLAY "Second number = " REAL-1.
    MOVE "Thank you" TO STRING-2.
    EXIT PROGRAM.
```

Terminal Dialog (assumes FORTRAN source is on file FPROG, and COBOL source is on file CSUB):

`/fortran input=fprog binary_object=fbin`	Compile FORTRAN program.
`/cobol input=csub binary_object=cbin sp=true`	Compile COBOL subprogram.
`/execute_task (fbin, cbin)`	Execute program.

```
Here are some numbers
First number =                    4
Second number =    +.24140000000000E+001
Thank you
```

Program output.

Calling C Routines

You can call a C routine from a FORTRAN program using a standard CALL statement or function reference. Because of different calling sequences and naming conventions, you must first declare the name of the C routine in a C$ EXTERNAL compiler directive (described in chapter 11). The C$ EXTERNAL compiler directive allows your program to reference an external routine by a valid FORTRAN program unit name. Values can be passed between the FORTRAN program and C routine through the argument list.

The arguments passed to the C routine can be constants, symbolic constants, variables, expressions with operators, or arrays. Arguments can be of type integer, real, or character. In C, character data are considered to be arrays of single characters.

All C strings should be terminated by a binary zero (ASCII NULL) character. Character strings passed to C should have a zero character at the end of the string. The receiving C program treats them as a pointer to an array of characters.

The storage sequence for C arrays having dimension greater than one differs from that of FORTRAN. In FORTRAN, arrays are stored in columnwise order, whereas in C, arrays are stored in rowwise order. Thus, for example, the corresponding locations of a FORTRAN array dimensioned (3,2) and a C array dimensioned (3,2) are as follows:

(1,1)	(2,1)	(3,1)	(1,2)	(2,2)	(3,2)	FORTRAN array
(1,1)	(1,2)	(2,1)	(2,2)	(3,1)	(3,2)	C array

If a character variable of length 3 (plus 1 for the zero) is passed to a C routine, it is treated as a character array of size 3, with each array element representing a single character:

FORTRAN Character Data	C Character Data
CHARACTER COLOR*4 COLOR='RED'	char color[4]
The variable COLOR contains the character string 'RED '	The array color has the following values: color [0] = 'R' color [1] = 'E' color [2] = 'D' color [3] = 0

The correspondence of data types between FORTRAN and C is as follows:

FORTRAN Data Set	C Character Data
REAL	float or double
INTEGER	int
CHARACTER*N	char array[n]
DOUBLE PRECISION	-
COMPLEX	-
BOOLEAN	-
LOGICAL	-

Variables, constants, symbolic constants, and expressions that are passed to a C routine are normally passed by value, that is, the value of the parameter is passed rather than its address. You can pass variables, constants, symbolic constants, and expressions by address using the PTR function. Arrays are passed to a C routine by address; the address of the first array element is passed rather than its value.

Only integer and real function values are supported by C.

If a routine written in C expects an address of a data value, rather than the value itself, use the PTR intrinsic function with the data value name as an argument in the CALL statement or function reference. PTR(a) is a generic function that returns the address of a. The result of the PTR function can not be used within a FORTRAN program unit. (See chapter 9 for more information on intrinsic functions.)

Before using the C compiler or executing a routine written in C, you must execute these two NOS/VE commands (in either order):

```
/$system.c.setup
/set_working_catalog $user
```

NOTE

If the C routine you are calling performs input/output operations, you must call C_ioinit to set up the runtime I/O for C and you must call C_iofinalize to wrap up I/O and flush output when you are done. The names are case sensitive and must be written exactly as they are in the C routine.

The following example shows a FORTRAN main program that calls a C routine named see_me:

FORTRAN program:

```
      PROGRAM MEETING
      INTEGER IHOUR
      CHARACTER WEEKDAY*7
      IHOUR=5
C$    EXTERNAL (ALIAS='see_me', LANG=C) CPROC
      CALL CPROC (WEEKDAY)
      PRINT *, WEEKDAY, ' AT ', IHOUR
      END
```

C routine:

```
see_me (weekday)
  char weekday[7];
{   weekday[0] = 'T';
    weekday[1] = 'U';
    weekday[2] = 'E';
    weekday[3] = 'S';
    weekday[4] = 'D';
    weekday[5] = 'A';
    weekday[6] = 'Y';
}
```

Terminal dialog (assumes the FORTRAN source is in file MEETING_PROGRAM and the C source in in file weekday_c):

```
/fortran input=meeting_program binary_object=fbin
/c input=weekday_c binary=cbin
/execute task (fbin, cbin)
TUESDAY AT 5
```

The argument WEEKDAY is passed by value to the C routine. The character variable WEEKDAY of size 7 is treated as an array of size 7 in the C routine. The value of the first character of WEEKDAY is the same as the value of the first element of WEEKDAY and so forth.

Example:

```
      PROGRAM P
      INTEGER A
      PRINT *, 'A is ', A
C$    EXTERNAL (ALIAS='c_routine', LANG=C) CSUB
      CALL CSUB(PTR(A))
      PRINT *, 'A is now ', A
      END

      c_routine (pj)
      int *pj;
      {
      int a=10;
      *pj=a;
      }
```

The main program calls the C routine and passes the address of variable A. The C routine assigns a value to the address of A.

Terminal dialog (assumes FORTRAN source is in file P and the C routine is in file ptrprog_c):

```
/fortran input=p binary_object=fbin
/c input=ptrprog_c binarty=cbin
/execute_task (fbin, cbin)
A is 0
A is now 10
```

NOTE

Arithmetic values returned to a FORTRAN program unit must be word aligned, that is, begin on a multiple of 8 bytes. Byte, 2-byte, and 4-byte, integer values are not permitted.

Introduction to Debug E

Debug is an SCL command utility that lets you debug a program during execution. Using Debug, you can stop execution at selected points, display the values of selected variables, and resume execution.

Debug is easy to use. It requires no modification of your source code and no knowledge of assembly language. You can reference variables by their symbolic names rather than their addresses in memory. Furthermore, you do not need to interpret memory dumps, insert PRINT statements into your program, or use a load map.

Debug also allows you to create an abort file that contains Debug commands. These commands are executed when an execution error occurs. This feature can eliminate the need to reproduce the error in order to debug the program.

Debug can be used in line mode or screen mode. Also, you can use Debug to perform machine-level debugging as well as symbolic debugging. This discussion focuses on using Debug in screen mode for symbolic debugging. For information about line debugging, machine-level debugging, and other Debug features, see the Debug Usage manual.

Screen debugging gives you all of the Debug features with the ease of use of a full screen interface. You can execute Debug commands by pressing function keys rather than typing commands. Online help enables you to learn screen debugging as you use it.

Using Debug in screen mode, you can:

- View your source code as it executes (an arrow points to the next line to be executed).
- Change the values of program variables while execution is suspended.
- Change the location where execution of your program resumes.
- View the program units of your program.

Using Debug

Using Debug in screen mode requires that your terminal support screen operation. If your terminal is not set up for full screen operation, see the NOS/VE System Usage manual for terminal definitions that support the screen interface.

To execute your FORTRAN program with Debug and use the symbolic debugging capability, you must:

- Set the interaction style of your terminal to screen by entering the NOS/VE command CHANGE_INTERACTION_STYLE.
- Compile your program with the OPTIMIZATION_LEVEL=DEBUG and DEBUG_AIDS=ALL parameter specified.
- Execute your program in Debug mode. To do this, specify the DEBUG_MODE=ON parameter on the EXECUTE_TASK command.

For example, to prepare the source program EXAFORT contained in permanent file $USER.EXAMPLE_FORT for use with Debug, enter the following commands:

```
/vfortran input=$user.example_fort binary_object=lgo optimization_level=debug ..
../debug_aids=all
```

To execute EXAFORT to use Debug in screen mode, enter the following command:

```
/change_interaction_style style=screen
/execute_task file=lgo debug_mode=on
```

The source listing of EXAFORT is displayed as follows on a CDC 721 terminal (on other terminals, the screen format may vary slightly).

```
(1)
(2)
     Debugging EXAFORT
     --->     PROGRAM EXAFORT

              CHARACTER TABLE(6)*3, LIST*18
(3)           REAL DIVDEND, DIVISOR, QUOTENT, COUNTER, RESULT
              INTEGER COLUMN, ROW
              DATA DIVDEND, DIVISOR, COLUMN/100.,0.,1/
              DATA LIST/'JANFEBMARAPRMAYJUN'/
     *****************************************************
     * TEST1:  Add to counter and call procedure to square and
     *         display count.
     *****************************************************
              DO 10 COUNTER = 1,10
                 CALL SQUARE (COUNTER)
           10 CONTINUE
     ------------------------- OUTPUT -------------------------
(4)                -- Welcome to Full Screen Debugging --

                          Press HELP for Assistance

(5) f1 StepN  f2 MSpeed  f3 Locate  f4 ChaVal  f5 DelBrk  f6 Deas  f7 ZmOut  f8 Opts
       Step1              HSpeed     SeeVal     SetBrk     Quit     Trace     Goto
```

(1)	Home line	The line on which you enter Debug commands and NOS/VE commands.
(2)	Message line	The line on which short messages from Debug are displayed.
(3)	Source window	The are in which the program you are debugging is displayed.
(4)	Output window	The area in which the output generated by your program (or output delivered by Debug) is displayed.
(5)	Row of functions	The Debug functions assigned to function keys.

How to Get Help

There are two ways to get help information while using Debug in screen mode:

1. The Help function.

   Using the Help function displays the Help window. The Help window overlays a portion of your screen and prompts you to enter the function for which you need help. If you press a function key, a short description of the function you select is displayed in the Help window. To exit Help, press RETURN. Upon exiting Help, your screen is restored to its original contents.

2. The HELP command.

   You can request help by entering the HELP command on the home line. This command is used to read an online manual while you are debugging your program. To leave the online manual, press Quit. When you leave the online manual, the screen is restored to its contents before you entered HELP. For example, if you need information about FORTRAN constants, press the HOME key to move the cursor to the home line and type the following HELP command on the home line:

   ```
   help s=constants m=fortran
   ```

   This command takes you to the FORTRAN online manual for an explanation of FORTRAN constants. To return to debugging, press Quit. See the NOS/VE System Usage manual for more information about the HELP command.

Screen Debug Example

This example demonstrates some commonly used Debug functions. It is represented as a series of steps. To get the most benefit from this example, you should create the sample program, EXAFORT, illustrated in figure K-1, and then perform each step.

EXAFORT is divided into the following test cases:

TEST1

A loop that increments a counter and then calls a subprogram to square and display the count. TEST1 demonstrates the use of the ChaVal, Goto, HSpeed, SeeVal, Step1, and StepN functions.

TEST2

A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table and displays each entry. TEST2 shows how to step through loops, use Debug line commands in screen mode, and how to scroll through Debug and program output data.

TEST3

A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

In this example, you will learn how to:

- Prepare and execute program EXAFORT to use Debug.
- Display screen functions.
- Set breaks.
- Debug test case TEST1.
- Debug test case TEST2.
- Debug test case TEST3.

In each test case, the application of some Debug functions is demonstrated. After you work this example, you can begin to debug your FORTRAN programs using Debug in screen mode.

```
      PROGRAM EXAFORT

      CHARACTER TABLE(6)*3, LIST*18
      REAL DIVDEND, DIVISOR, QUOTENT, COUNTER, RESULT
      INTEGER COLUMN, ROW
      DATA DIVDEND, DIVISOR, COLUMN/100.,0.,1/
      DATA LIST/'JANFEBMARAPRMAYJUN'/

**************************************************************
*  TEST1:  Add to counter and call procedure to square and   *
*          display count.                                    *
**************************************************************

      DO 10 COUNTER = 1,10
        CALL SQUARE (COUNTER)
   10 CONTINUE

**************************************************************
*  TEST2:  Create single column table for each month.        *
**************************************************************

      DO 20 ROW = 1,6
        TABLE(ROW) = LIST(COLUMN : COLUMN + 2)
        PRINT*, 'THE MONTH IS:  ', TABLE(ROW)
        COLUMN = COLUMN + 3
   20 CONTINUE

**************************************************************
*  TEST3:  Create divide fault.                              *
**************************************************************

      QUOTENT = DIVDEND / DIVISOR
      PRINT*, 'ANSWER IS:  ', QUOTENT

      END

**************************************************************
*  Subroutine SQUARE                                         *
**************************************************************

      SUBROUTINE SQUARE (COUNTER)
        RESULT = 0.
        RESULT = COUNTER * COUNTER
        PRINT*, COUNTER, ' TIMES ', COUNTER, ' = ', RESULT
      END
```

Figure E-1. Example of EXAFORT Source Listing

Preparing and Executing Program EXAFORT to Use Debug

After you create program EXAFORT, you must:

1. Set the interaction style of your terminal screen.

2. Compile your program with special parameters to use Debug.

3. Turn on Debug mode.

4. Begin execution of your program. (This begins the Debug session.)

Do this as follows:

1. Set the interaction style of your terminal to screen by entering the following command:

   ```
   /vfortran input=$user.example_fort binary_object=lgo ..
   ../optimization_level=debug debug_aids=all
   ```

2. Assuming program EXAFORT is contained in permanent file $USER.EXAMPLE_FORT, compile EXAFORT by entering the following commands:

   ```
   /change_interaction_style style=screen
   ```

3. Enter the following command and turn on Debug mode to execute program EXAFORT:

   ```
   /execute_task file=lgo debug_mode=on
   ```

The Debug session begins and the source listing of EXAFORT is displayed in the Source window. The menu of Debug functions is displayed at the bottom of the screen.

Displaying Screen Functions

At the beginning of a Debug session, you can use the following function to display helpful information about the debugging environment:

`Help`

Displays Help information.

Perform the following steps to become familiar with the Debug functions:

1. Press the `Help` function key. The Help window is displayed.

2. Press each function key corresponding to a function displayed in the menu of functions. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window. If you press `Help` again, additional text describing the function from the Debug online manual is displayed.

3. Press RETURN. This exits Help.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks:

`Bkw`

Scrolls backward to the previous screen of text.

`First`

Displays the first screen of the source listing. Because `First` is a lower priority function, it may not be assigned to a function key on terminals with only 16 function keys. Instead, FIRST must be entered on the home line.

`Fwd`

Scrolls forward to the next screen of text.

`Locate`

Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.

`SetBrk`

Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the statement on the line containing the break.

Perform the following steps to place three execution breaks in EXAFORT:

1. Press the `Locate` function key. At the top right-hand corner of the screen, you are prompted for the text to be located.

2. Enter the following text exactly as it appears in EXAFORT:

   ```
   DO 20
   ```

   The cursor is moved to the line:

   ```
   DO 20 ROW = 1,6
   ```

3. Press the `SetBrk` function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.

4. Use the down-arrow key to move the cursor to the line containing:

   ```
   COLUMN = COLUMN + 3
   ```

   If you do not see this line on your screen, use the `Fwd` function key. The next screen of the EXAFORT source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

5. Press the `SetBrk` function key. The line is highlighted to show that it contains an execution break.

6. Use the down-arrow key to move the cursor to the line:

```
QUOTENT = DIVDEND / DIVISOR
```

If you do not see this line on your screen, use the `Fwd` function key. The next screen of the EXAFORT source listing is displayed. Use the down-arrow key to position the cursor on the correct line.

7. Press the `SetBrk` function key. The line is highlighted to show that it contains an execution break.

8. Press the `First` function key. The first screen of the EXAFORT source listing is displayed in the source window.

If `First` is not assigned to a function key, FIRST must be entered on the home line. To do this, use the HOME key. This moves the cursor to the home line. Enter the following on the home line:

```
first
```

The first screen of the EXAFORT source listing is displayed in the source window.

Debugging TEST1

Using Debug, you can execute a program one line or several lines at a time. Also, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

`ChaVal`

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

`Goto`

Moves the execution arrow to the line that contains the cursor. Execution resumes with the statement on this line.

`HSpeed`

Executes a program until a break is encountered or the program ends.

`SeeVal`

Prompts you to enter a variable name, then displays the value of the variable in the output window.

`Step1`

Executes a program one line at a time.

`StepN`

Executes N lines of a program, where N is an integer.

Perform the following steps to demonstrate the use of the `ChaVal`, `Goto`, `HSpeed`, `SeeVal`, `Step1`, and `StepN` functions:

1. Press the `Step1` function key. The first statement of EXAFORT is executed, moving the execution arrow to the statement:

   ```
   DO 10 COUNTER = 1,10
   ```

2. Press the `Step1` function key again. The DO statement is executed; the execution arrow points to the statement:

   ```
   CALL SQUARE (COUNTER)
   ```

3. Press the `Step1` function key six times. An iteration of TEST1 is executed one line at a time. The output from the iteration is displayed in the Output window.

4. Press the `SeeVal` function key. A prompt to enter a variable name is printed in the upper right-hand corner of the screen. Enter the name:

   ```
   counter
   ```

   The value of COUNTER is displayed in the Output window:

   ```
   counter = 2.
   ```

   Thus, you can use `SeeVal` to observe the contents of a variable.

5. Press the ChaVal function key. A prompt for a variable name and its new value is displayed in the upper right-hand corner of the screen; enter:

   ```
   counter=8
   ```

   The value of COUNTER is changed to 8.

6. Press the SeeVal function key. When you are prompted for a variable name, enter:

   ```
   counter
   ```

   The following message is displayed in the Output window:

   ```
   counter = 8.
   ```

   Thus, the change of COUNTER's value is verified.

7. Press the StepN function key. In the upper right-hand corner of the screen, you are prompted for the number of lines to execute; enter:

   ```
   6
   ```

   StepN executes 6 lines of TEST1. The output from this loop iteration is displayed in the Output window.

8. Press the SeeVal function key. When you are prompted for a variable name, enter:

   ```
   counter
   ```

   The value of COUNTER is displayed in the Output window:

   ```
   counter = 9.
   ```

   Therefore, the value given to COUNTER in step 5 is used by the DO statement.

9. Use the up-arrow key to move the cursor to the line:

   ```
   DO 10 COUNTER = 1,10
   ```

10. Press the Goto function key. The execution arrow moves to the line containing the cursor; execution resumes with this statement.

11. Press the HSpeed function key. Execution resumes from the DO statement; COUNTER is initialized to 1. Execution of EXAFORT continues until an execution break is encountered.

Debugging TEST2

After program execution is resumed in step 11 of TEST1, it stops at the break set on the DO statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

`Bkw`

Scrolls backward to the previous screen of text.

`DelBrk`

Deletes execution breaks.

`HSpeed`

Executes a program until a break is encountered or the program ends.

This section also uses the following items:

HOME

Press the HOME key to move the cursor to the home line. Debug line commands can be entered on the home line for execution in screen mode.

DISPLAY_PROGRAM_VALUE

A Debug line command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute Debug line commands, and scroll output data:

1. Press the `HSpeed` function key. Execution stops at the break set on the last line of the DO loop in TEST2; output from the loop is displayed in the Output window.

2. Press the `HSpeed` function key again. One iteration of the DO loop is executed; execution stops at the break set at the statement, COLUMN = COLUMN + 3. Each time `HSpeed` is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the `HSpeed` function key. One more loop iteration is performed.

4. Press the HOME key. The cursor moves to the home line.

5. Enter the Debug line command:

   ```
   display_program_value name=$all
   ```

   The values of all variables in EXAFORT are displayed in the Output window. Thus, Debug line commands can be used in screen mode by entering them on the home line. For more information about using Debug line commands, see the Debug Usage manual.

6. Press the DelBrk function key. The execution break is deleted.

7. Press the down-arrow key until the cursor is inside of the Output window.

8. Press the Bkw function key. The data in the Output window scrolls backward. When the cursor is contained within the Output window, you can use the Bkw and Fwd functions to scroll backward and forward through the data in the window.

9. Press the HSpeed function key. The execution of EXAFORT resumes, stopping when the line containing the third break is reached. The execution arrow points to the first statement of TEST3.

Debugging TEST3

After resuming execution of EXAFORT in step 9 of section TEST2, execution stops at the beginning of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an execution error is encountered:

`ChaVal`

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

`Goto`

Moves the execution arrow to the line that contains the cursor. Execution resumes with the statement on this line.

`SeeVal`

Prompts you to enter a variable name, then displays the value of the variable in the Output window.

`Step1`

Executes a program one line at a time.

`Quit`

Used to exit Debug.

Perform the following steps to finish the example:

1. Press the `Step1` function key. The DIVISION statement is executed, execution of EXAFORT halts, and the following message flashes in the top right hand corner of the screen:

   ```
   divide_fault
   ```

2. Press the `SeeVal` function key. When you are prompted for a variable name, enter:

   ```
   divisor
   ```

   The following message is displayed in the Output window:

   ```
   divisor = 0.
   ```

   A division by zero caused the execution error.

3. Press the `ChaVal` function key. When you are prompted, enter:

   ```
   divisor=1
   ```

   The value of DIVISOR is changed to 1.

4. Press the `SeeVal` function key. When you are prompted, enter:

   ```
   divisor
   ```

   The following text is displayed in the Output window:

   ```
   divisor = 1.
   ```

   The change to DIVISOR is verified.

5. Press the Goto function key. The execution arrow points at the DIVISION statement and program execution resumes with this statement.

6. Press the Step1 function key. The DIVISION statement is executed. Therefore, the Goto and ChaVal functions can be used together to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

7. Press the Step1 function key again. The result of the DIVISION statement is displayed in the Output window.

8. Press the Step1 function key. EXAFORT ends and the following message is displayed in the Output window:

```
DEBUG:  The status at termination was: NORMAL.
```

9. Press the Quit function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using Debug in screen mode to debug your FORTRAN programs. For more information about the Debug utility, see the Debug Usage manual.

The Programming Environment and the Professional Programming Environment F

The Programming Environment and Professional Programming Environment both offer an integrated screen interface to NOS/VE programming tools. The Programming Environment is designed for a one-person programming project; the Professional Programming Environment is designed for a multi-person programming project.

Programming Environment

The Programming Environment is a full screen utility that provides functions to facilitate programming in FORTRAN on NOS/VE. The Programming Environment for NOS/VE Summary/Tutorial (publication number 60486819) provides more details about the Programming Environment.

Entering the Environment

The Programming Environment can be entered using the command

ENTER_PROGRAMMING_ENVIRONMENT or
ENTPE
DEFAULT_PROCESSOR = keyword,
ENVIRONMENT_CATALOG = catalog_path

ENVIRONMENT_CATALOG must be the *catalog_path* that the environment is to use to store its files. This parameter defaults to $USER.PROGRAMMING_ENVIRONMENT.

DEFAULT_PROCESSOR may be FORTRAN, COBOL, PASCAL VECTOR_FORTRAN or C. VECTOR_FORTRAN selects the FORTRAN Version 2 programming language. This value is displayed in the programming language field of the environment screens, and is used to specify the processor for each program created. FORTRAN is the default value.

Do not modify or delete any files in the Programming Environment catalog. If the environment files are altered, Control Data cannot be responsible for the proper functioning of the environment.

Providing HELP Information

Depending on where you are in the Programming Environment, a HELP request can:

- Generate a short message
- Take you to a menu of HELP options
- Provide explanations of the current screen
- Take you to the programming language usage manual
- Provide an explanation of the functions currently available

Creating a Program

You can use the Create function to:

- Give the environment the name of a new program; then code that program while in the environment
- Specify an existing file, whose contents are to become a program, to be used in the environment

Modifying a Program

You can use the Modify function to:

- Enter the NOS/VE file editor and edit a program
- Call the usage manual of the current programming language
- Format the current program according to the formatting convention of the current programming language

Running a Program

You can use the Run function to:

- Compile a program
- Compile a program that has changed or has compilation parameters that have changed
- Run a previously compiled program
- Fix the detected compilation errors
- Get a message about the status of the run
- Alter runtime parameters

Debugging a Program

You can use the Debugging function to:

- Call up the Debug utility
- Set and delete breaks for debugging
- Run the program until the next interrupt
- Display a program value
- Change a program value
- Execute one line at a time
- Execute n lines at a time
- Terminate program execution

Printing Components

You can use the Printing function to print the following components:

- Source programs
- Compilation listings
- Terminal output
- Loadmap

Viewing Components

You can use the View function to view the following:

- Source programs
- Compilation listings
- Terminal output
- Performance graph
- Loadmap

Delete

You can use the Delete function to:

- Remove a component of a program to conserve file space
- Delete a program

Restore

You can use the Restore function to:

- Restore a deleted component within a Programming Environment session
- Restore a deleted program within a Programming Environment session

Exporting a Component

You can copy a component from the environment into a user-specified file.

Generating a Program Performance Graph

You can use this function to generate a graph of program performance showing:

- User written routines
- System routines
- Number of calls
- Amount of time spent in each procedure

Tailoring the Environment

You can use this function to change the FORTRAN, COBOL, Pascal, VECTOR_FORTRAN (for FORTRAN Version 2), or C program templates.

Changing Runtime Parameters

You can use this function to:

- Display the current program parameters
- Change the value of a program parameter
- Return a parameter to its compiler default setting
- View parameter lists stored in the parameter list library
- Save a parameter list in the library of named parameter lists

Import a Program

You can bring an existing source program into the environment with this function.

Printing Components

You can use the Printing function to print the following components:

- Source programs
- Compilation listings
- Terminal output
- Loadmap

Viewing Components

You can use the View function to view the following:

- Source programs
- Compilation listings
- Terminal output
- Performance graph
- Loadmap

Delete

You can use the Delete function to:

- Remove a component of a program to conserve file space
- Delete a program

Restore

You can use the Restore function to:

- Restore a deleted component within a Programming Environment session
- Restore a deleted program within a Programming Environment session

Exporting a Component

You can copy a component from the environment into a user-specified file.

Generating a Program Performance Graph

You can use this function to generate a graph of program performance showing:

- User written routines
- System routines
- Number of calls
- Amount of time spent in each procedure

Tailoring the Environment

You can use this function to change the FORTRAN, COBOL, Pascal, VECTOR_FORTRAN (for FORTRAN Version 2), or C program templates.

Changing Runtime Parameters

You can use this function to:

- Display the current program parameters
- Change the value of a program parameter
- Return a parameter to its compiler default setting
- View parameter lists stored in the parameter list library
- Save a parameter list in the library of named parameter lists

Import a Program

You can bring an existing source program into the environment with this function.

Printing Components

You can use the Printing function to print the following components:

- Source programs
- Compilation listings
- Terminal output
- Loadmap

Viewing Components

You can use the View function to view the following:

- Source programs
- Compilation listings
- Terminal output
- Performance graph
- Loadmap

Delete

You can use the Delete function to:

- Remove a component of a program to conserve file space
- Delete a program

Restore

You can use the Restore function to:

- Restore a deleted component within a Programming Environment session
- Restore a deleted program within a Programming Environment session

Exporting a Component

You can copy a component from the environment into a user-specified file.

Generating a Program Performance Graph

You can use this function to generate a graph of program performance showing:

- User written routines
- System routines
- Number of calls
- Amount of time spent in each procedure

Tailoring the Environment

You can use this function to change the FORTRAN, COBOL, Pascal, VECTOR_FORTRAN (for FORTRAN Version 2), or C program templates.

Changing Runtime Parameters

You can use this function to:

- Display the current program parameters
- Change the value of a program parameter
- Return a parameter to its compiler default setting
- View parameter lists stored in the parameter list library
- Save a parameter list in the library of named parameter lists

Import a Program

You can bring an existing source program into the environment with this function.

Professional Programming Environment

The Professional Programming Environment (PPE) is primarily for users working as part of a multi-person programming project that is developing a product for use under the NOS/VE operating system.

Entering PPE

After preparing your terminal for full-screen use (see the Professional Programming Environment manual for information on this), you can start PPE by using the SCL command:

ENTER_PPE or
ENTP
ENVIRONMENT_CATALOG=catalog_path
STATUS=status variable

ENVIRONMENT_CATALOG or *EC*

Path to the subcatalog for which PPE is executed. It is the lowest level of the PPE hierarchy presented in the session.

PPE creates the subcatalog if it does not exist. If the subcatalog belongs to another user, the owner must grant you the following catalog permit:

Access_Modes=(all, cycle, control)

Application_Information='I1'

If you omit the ENVIRONMENT_CATALOG parameter, the subcatalog used is $USER.PROFESSIONAL_ENVIRONMENT.

STATUS

Optional status variable in which the command returns its completion status.

PPE Capabilities

As a programming environment, PPE integrates the programming tasks, including:

- Editing source text.
- Compiling source text.
- Debugging source text.
- Executing object code.

In addition, PPE can coordinate the activities of a multi-person programming project providing these capabilities:

- Full-screen interface to SCU deck and modification creation.
- Extraction and transmittal of SCU decks and modifications within a source library hierarchy. It enforces interlocks to ensure that only one copy of a deck can be changed.
- Expansion and compilation of the product source, including copying decks from higher levels of the hierarchy when a deck is not present at the lower level.

- Tracing of compilation errors to the decks containing the source.
- Maintenance of an object library at each level of the hierarchy. Each object library contains the compiled code for the source decks at that level.
- Partial builds of the product, expanding and compiling only those source decks that have changed.
- Execution of the product version at the current level of the hierarchy, using object modules at higher levels as needed.

PPE Limitations

- All code for the product being developed must be written in one language.
- The programming languages supported are NOS/VE FORTRAN Version 1, FORTRAN Version 2, CYBIL, and COBOL.
- PPE does not provide a method of using SCU selection criteria files.

Control Data Extensions to Standard FORTRAN G

Following is a list of the FORTRAN Version 2 features that are extensions and additions to ANSI FORTRAN.

- Basic Concepts
  - Symbolic names can be up to 31 characters in length (ANSI allows only six) and can contain underscores or dollar signs ($).
  - The " (quote), ! (exclamation point), and _ (underscore) characters have been added to the FORTRAN character set.
  - C$ (compiler control) directives have been added.
  - A boolean data type has been added.
  - Inline comments have been added.
  - A byte data type has been added.
- Constants
  - Boolean constants have been added (the boolean constants are boolean string, octal, and hexadecimal).
  - Extended Hollerith constants have been added.
  - Symbolic constants can appear as the real and imaginary parts of a complex constant.
  - Byte symbolic constants have been added.
  - Two- and four-byte integer symbolic constants have been added.
  - Sixteen-byte real symbolic constants have been added.
- Arrays and Substrings
  - Intrinsic function references and boolean constants can appear in dimension bound expressions.
  - Subscript or substring expressions can be real, double precision, complex, or boolean expressions, as well as integer.
  - Array section references.
  - Array valued intrinsic functions.

- Expressions
  - Boolean expressions have been added.
  - Boolean expressions can appear in arithmetic expressions.
  - Double precision and complex operands can be combined using the +, -, *, and / operators.
  - A double precision operand can be raised to a complex power.
  - Boolean entities can appear in relational expressions.
  - An .XOR. operator has been added.
  - Constant expressions can include intrinsic function references with constant expressions as arguments.
  - Integer expressions can contain byte and two- and four-byte integer values.
  - Real expressions can contain 16-byte real values.
- Specification Statements
  - A BOOLEAN type statement has been added.
  - A BYTE type statement has been added.
  - Entities in named (labeled) common can be initialized by a DATA statement in any program unit.
  - Extensible common has been added.
  - Mixed common is allowed.
  - The IMPLICIT statement can declare a boolean or byte type.
  - The IMPLICIT NONE statement has been added.
  - The PARAMETER statement can declare boolean or byte symbolic constants.
  - A symbolic constant can appear as the real or imaginary part of a complex constant.
  - The INTEGER*length (where length=2, 4, or 8) statement has been added.
  - The REAL*length (where length=8 or 16) statement has been added.
  - The INTERFACE and ENDINTERFACE statements have been added.
- DATA Statement
  - A replicated value list can appear in a DATA statement.
  - Boolean and byte entities can appear in a DATA statement.

- Assignment Statement
  - Assignment statements can be type boolean.
  - A multiple assignment statement has been added.
  - An assignment statement can contain array-valued entities.
- Flow Control Statements
  - Real, double precision, complex, or boolean expressions are valid in a computed GO TO statement.
  - A boolean expression can be used in an arithmetic IF statement.
  - A boolean expression can be used as an indexing parameter in a DO loop or implied DO list.
  - A one-trip DO loop option has been added for increased compilation speed.
  - An extended range capability for DO loops has been added.
  - The WHERE statement has been added.
- Input/Output
  - The following input/output statements have been added:
    - NAMELIST
    - BUFFER IN and BUFFER OUT
    - ENCODE and DECODE
    - PUNCH
    - OPENMS, READMS, WRITMS, CLOSMS, STINDX
  - A record length specifier can appear in an OPEN statement for a file accessed sequentially.
  - More than one unit can be associated with a single external file.
  - Random access files have been added.
  - Segment access files have been added.
  - Extended internal files have been added.
  - An external unit identifier can be type boolean.
  - A buffer length specifier can appear in an OPEN statement (this specifier is disregarded in Control Data FORTRAN).
  - A comma can optionally follow the output list of a list directed output statement.

- An implicit file/unit association occurs in the absence of PROGRAM statement or OPEN statement declaration.
- The following edit descriptors have been added:
  - Quoted string (" ... ")
  - Rw (character)
  - Ow and Ow.m (octal)
  - Zw and Zw.m (hexadecimal)
- NAMELIST formatting has been added.
- A and Aw descriptors can be used for noncharacter data.
- A format specification can be contained in a noncharacter array.
- A FORM='BUFFERED' option has been added to the OPEN and INQUIRE statements.
- File references can appear for file names in input/output statements.

• PROGRAM Statement

- Symbolic unit specifiers can be declared on the PROGRAM statement.
- Buffer length (disregarded by FORTRAN) and record length can be declared on the PROGRAM statement.
- In a statement function reference, the actual argument is converted to the type of the dummy argument.
- A substring reference can appear in the expression of a statement function statement.

• External Procedures

- Boolean arguments can be associated with integer or real arguments.
- An external procedure name can be the same as a common block name.
- The name of a block data subprogram can be the same as a common block name.
- A RETURN statement can appear in a main program (it has the same effect as an END statement).
- The expression in the alternate return form of the RETURN statement can be any arithmetic or boolean expression. (ANSI allows only integer expressions.)
- Extended Hollerith constants can be passed as actual arguments.

- Intrinsic Functions
  - The following intrinsic functions have been added:
    - BOOL
    - array processing (ALL, ALLOCATED, ALT, ANY, COUNT, DOTPRODUCT, LBOUND, MATMUL, PACK, RANK, SEQ, SHAPE, SIZE, SUM, UBOUND, UNPACK)
    - boolean operations (AND, OR, XOR, NEQV, EQV, COMPL)
    - mathematical (ERF, ERFC, ATANH, SIND, COSD, TAND, COTAN)
    - miscellaneous (SHIFT, MASK, RANF, EXTB, INSB, SUM1S, MERGE, PTR)
  - ALL functions can have boolean arguments.
- Association of Entities
  - Partial association can exist between a boolean entity and a double precision entity.
  - Association can exist between a boolean entity and an integer or real entity.
- FORTRAN-Callable Subprograms
  - The following subprograms can be called from a FORTRAN program:
  - Keyed-File Interface subprograms
  - Sort/Merge subprograms
  - System Command Language subprograms
  - Utility subprograms
  - Input/Output status checking subprograms
  - Miscellaneous input/output subprograms
  - Debugging subprograms
  - Collating sequence control subprograms
  - NOS/VE status subprograms

Language Summary H

The following symbols are used in the descriptions of the FORTRAN statements:

v	variable name, array name, or array element
sl	statement label
iv	integer variable
name	symbolic name
u	input/output unit specifier, which can be an integer expression with a value of 0 through 999, or a boolean expression whose value is a unit name in L format
fs	format specification
iolist	input/output list
ios	input/output status indicator
recn	record number
length	length, in bytes, of an entity. Options for arithmetic entities are 2, 4, 8, or 16, depending on the type. Options for character entities are 1 through 256.

Other symbols are defined individually in the statement descriptions.

Assignment

arithmetic v = arithmetic or boolean expression

boolean v = boolean or arithmetic expression

character v = character expression

logical v = logical or relational expression

Multiple Assignment

v=...*v*=expression

Type Declaration

INTEGER*length* **v***length*, ...,*v*length*

BYTE v, ...,v

REAL*length* **v***length*, ...,*v*length*

DOUBLE PRECISION v, ...,v

COMPLEX*length* **v***length*, ...,*v*length*

BOOLEAN v, ...,v

LOGICAL*length* **v***length*, ...,*v*length*

CHARACTER*length* **v***length*,...,*v*length*

IMPLICIT type(ac,...,*ac*),...,*type(ac,...,ac)*

ac

A single letter, or range of letters represented by the first and last letter separated by a hyphen, indicating which variables are implicitly typed.

IMPLICIT NONE

External Declaration

EXTERNAL name,...,*name*

Intrinsic Declaration

INTRINSIC name*length*,...,*name *length*

Storage Allocation

type array(d),...,*array(d)*

type

Is INTEGER**length*, CHARACTER**length*, BOOLEAN, BYTE, REAL**length*, COMPLEX**length*, DOUBLE PRECISION, or LOGICAL**length*.

d

A dimension bound expression. You can have one through 7 dimensions, as described in chapter 4 under Declaring Arrays.

ALLOCATE(ad,...,*ad*)

ad

Is a constant or adjustable array declaration.

DEALLOCATE(a,...,*a*)

a

Is a name of an allocatable array.

DIMENSION array(d),...,*array(d)*)

d

A dimension bound expression. You can have one through 7 dimensions, as described in chapter 4 under Declaring Arrays.

COMMON */name/***nlist**,...,*/name/nlist*

nlist

Is a list of variables or arrays, separated by commas, to be included in the common block.

DATA nlist/clist/,...,*nlist/clist/*

nlist

Is a list of names to be initially defined. Each name in the list can take the form:

variable
array
element
substring
implied DO list

clist

Is a list of constants or symbolic constants specifying the initial values. Forms for list items are described in chapter 4 under the DATA statement.

EQUIVALENCE (nlist),...,*(nlist)*

nlist

Is a list of variable names, array names, array element names, or character substring names. The names are separated by commas.

PARAMETER (name=exp,...,*name=exp*)

exp

Is an extended expression.

SAVE *name,...,name*

Storage Allocation

type array(d),...,*array(d)*

type

Is INTEGER**length*, CHARACTER**length*, BOOLEAN, BYTE, REAL**length*, COMPLEX**length*, DOUBLE PRECISION, or LOGICAL**length*.

d

A dimension bound expression. You can have one through 7 dimensions, as described in chapter 4 under Declaring Arrays.

ALLOCATE(ad,...,*ad*)

ad

Is a constant or adjustable array declaration.

DEALLOCATE(a,...,*a*)

a

Is a name of an allocatable array.

DIMENSION array(d),...,*array(d)*)

d

A dimension bound expression. You can have one through 7 dimensions, as described in chapter 4 under Declaring Arrays.

COMMON */name/***nlist**,...,*/name/nlist*

nlist

Is a list of variables or arrays, separated by commas, to be included in the common block.

DATA nlist/clist/,...,*nlist/clist/*

nlist

Is a list of names to be initially defined. Each name in the list can take the form:

variable
array
element
substring
implied DO list

clist

Is a list of constants or symbolic constants specifying the initial values. Forms for list items are described in chapter 4 under the DATA statement.

EQUIVALENCE (nlist),...,*(nlist)*

nlist

Is a list of variable names, array names, array element names, or character substring names. The names are separated by commas.

PARAMETER (name=exp,...,*name=exp*)

exp

Is an extended expression.

SAVE *name,...,name*

Flow Control

GO TO sl

GO TO (sl,...,*sl*) expression

GO TO iv,*(sl,...,sl)*

ASSIGN sl TO iv

IF (arithmetic or boolean expression) **sl_1,sl_2,sl_3**

IF (logical expression) **statement**

IF (logical expression) **THEN**

ELSE IF (logical expression) **THEN**

ELSE

END IF

WHERE (logical array expression) **array assignment statement**

WHERE (logical array expression)

ELSEWHERE

ENDWHERE

DO sl,v=e_1,e_2,e_3

e_1,e_2,e_3
Are indexing parameters. They can be integer (any length), byte, real, double precision, or boolean constants, symbolic constants, variables, or expressions.

CONTINUE

PAUSE *n*

n
A string of 1 through 5 digits, or a character constant of 1 through 70 characters

STOP *n*

n
A string of 1 through 5 digits, or a character constant of 1 through 70 characters

END

Main Program

PROGRAM *name* **(upar,...,***upar***)**

upar

A unit declaration in one of the following forms: *unitname*

unitname=buffer-length

unitname=record-length

unitname=buffer-length/record-length

alternate-name=unitname

(buffer-length is disregarded under NOS/VE)

Subprogram

SUBROUTINE name *(argument,...,argument)*

type **FUNCTION name****length(argument,...,argument)*

type

is BOOLEAN, CHARACTER, INTEGER**length*, BYTE, REAL**length*, COMPLEX**length*, DOUBLE PRECISION, or LOGICAL**length*.

BLOCK DATA *name*

Statement Function

name *(argument,...,argument)***=expression**

Subroutine Call

CALL name *(argument,...,argument)*

Function Reference

name *(argument,...,argument)*

Entry Point

ENTRY name *(argument,...,argument)*

Return

RETURN *arithmetic or boolean expression*

Formatted Input/Output

READ (*UNIT*=**u**, *FMT*=**fs**, *IOSTAT=ios, ERR=sl, END=sl) iolist*

READ fs,*iolist*

WRITE (*UNIT*=**u**, *FMT*=**fs**, *IOSTAT=ios, ERR=sl) iolist*

PRINT fs, *iolist*

PUNCH fs, *iolist*

Unformatted Input/Output

READ (*UNIT*=**u**, *IOSTAT=ios, ERR=sl, END=sl) iolist*

WRITE (*UNIT*=**u**, *IOSTAT=ios, ERR=sl) iolist*

List Directed Input/Output

READ (*UNIT*=**u**, *FMT*=*, *IOSTAT=ios, ERR=sl, END=sl) iolist*

READ *, iolist

WRITE (*UNIT*=**u**, *FMT*=*, *IOSTAT=ios, ERR=sl) iolist*

PRINT *, iolist

PUNCH *, iolist

Direct Access Input/Output

READ (*UNIT*=**u**, *FMT*=**fs**, *IOSTAT*=*ios*, *ERR*=*sl*, **REC=recn**) *iolist*

WRITE (*UNIT*=**u**, *FMT*=**fs**, *IOSTAT*=*ios*, *ERR*=*sl*, **REC=recn**) *iolist*

Namelist Input/Output

NAMELIST /name/v,...,*v* ... */name/v*,...,*v*

READ (*UNIT*=**u**, *FMT*=**name**, *IOSTAT*=*ios*, *ERR*=*sl*, *END*=*sl*)

READ name

WRITE (*UNIT*=**u**, *FMT*=**name**, *IOSTAT*=*ios*, *ERR*=*sl*)

PRINT name

PUNCH name

name
A namelist group name.

Buffer Input/Output

BUFFER IN (u,p) (a,b)

BUFFER OUT (u,p) (a,b)

p

Disregarded under NOS/VE.

a

The first word of the data block to be transferred.

b

The last word of the data block to be transferred.

Internal Data Transfer

ENCODE (c,fs,v) iolist

DECODE (c,fs,v) iolist

v

The starting location of the record to be transferred.

c

Specifies the number of characters to be transferred to or from each record.

Format Specification

sl FORMAT (flist)

flist

Is a list of items, separated by commas, having the following forms:

red
ned
r(flist)

ed

Is a repeatable edit descriptor.

ned

Is a nonrepeatable edit descriptor.

r

Is a nonzero unsigned integer constant repeat specification.

Table H-1. Edit Descriptors

Format	Description
*sr***Ew.d**	Single precision floating-point with exponent.
*sr***Ew.dEe**	Single precision floating-point with specified exponent length.
*sr***Fw.d**	Single precision floating-point without exponent.
*sr***Dw.d**	Double precision floating-point with exponent.
*sr***Gw.d**	Single precision floating-point with or without exponent.
*sr***Gw.dEe**	Single precision floating-point with or without specified exponent length.
*r***Iw**	Decimal integer.
*r***Iw.m**	Decimal integer with specified minimum number of digits.
*r***Lw**	Logical.
*r***A**	Character with variable length.
*r***Aw**	Character with specified length.
*r***Rw**	Rightmost characters with binary zero fill.
*r***Ow**	Octal.
*r***Ow.m**	Octal with minimum digits and leading zeros.
*r***Zw**	Hexadecimal.
*r***Zw.m**	Hexadecimal with minimum digits and leading zeros.

(Continued)

Table H-1. Edit Descriptors *(Continued)*

Format	Description
kP	Changes the position of a decimal point of an input or output real number.
BN	Blanks ignored on numeric input.
BZ	Blanks treated as zeros on numeric input.
SP	+ characters produced on output.
SS	+ characters suppressed on output.
S	+ characters suppressed on output.
nX	Skip n spaces.
Tn	Tabulate to n^{th} column.
TRn	Tabulate forward.
TLn	Tabulate backward.
nH	Boolean or character string output.
"..."	Boolean or character string output.
'...'	Character string output.
:	Format control.
/	End of FORTRAN record.

s optional scale factor of the form kP.

r optional repetition factor.

w integer constant indicating field width.

d integer constant indicating digits to right of decimal point.

e integer constant indicating digits in exponent field.

m integer constant indicating minimum number of digits in field.

n positive nonzero decimal digit.

k integer constant called a scale factor.

File Positioning

BACKSPACE (*UNIT*=**u**, *IOSTAT=ios, ERR=sl*)

BACKSPACE u

REWIND (*UNIT*=**u**, *IOSTAT=ios, ERR=sl*)

REWIND u

ENDFILE (*UNIT*=**u**, *IOSTAT=ios, ERR=sl*)

ENDFILE u

File Status

OPEN (*UNIT*=**u**, *IOSTAT=ios, ERR=sl, FILE=fin, STATUS=sta, ACCESS=acc, FORM=fm, RECL=rl, BLANK=blnk, BUFL=bl*)

INQUIRE (unit-or-file, *IOSTAT=ios, ERR=sl, EXIST=ex, OPENED=od, NUMBER=num, NAMED=nmd, NAME=fn, ACCESS=acc, SEQUENTIAL=seq, DIRECT=dir, FORM=fm, FORMATTED=FMT, UNFORMATTED=unf, RECL=fcl, NEXTREC=nr, BLANK=blnk*)

where unit-or-file is one of the following:

UNIT=u

FILE=filename

CLOSE (*UNIT*=**u**, *IOSTAT=ios, ERR=sl, STATUS=sta, SIZE=n*)

Differences Between FORTRAN Version 1 and FORTRAN Version 2 — I

This appendix presents the differences between FORTRAN Version 1 and FORTRAN Version 2. This discussion can be used as an aid in converting programs from FORTRAN Version 1 to FORTRAN Version 2.

FORTRAN Version 1 and FORTRAN Version 2 are highly compatible. There are minor differences that can lead to programs exhibiting different behaviors or answers. The currently known set of differences, and some things you can do to make them work the same, are detailed here.

The first section describes general differences that are apparent at all levels of vectorization of the FORTRAN Version 2 program. The second section describes differences that are only apparent when the FORTRAN Version 2 program is compiled with high vectorization (VECTORIZATION_LEVEL = HIGH).

General Differences

The following paragraphs describe various differences between FORTRAN Version 1 and FORTRAN Version 2 that are not affected by level of vectorization of the FORTRAN Version 2 program.

Compilation Command

The compilation command is different for FORTRAN Version 2; the command is VECTOR_FORTRAN. It supports all compilation parameters for the FORTRAN compilation command except SEQUENCED_LINES. VECTOR_FORTRAN has two additional parameters related to vector programming:

VECTORIZATION_LEVEL
REPORT_OPTIONS

Also, the OPTIMIZATION_OPTIONS parameter on the VECTOR_FORTRAN command has an additional option, ALTERNATIVE_ CODE_SELECTION. This option is used to optimize programs on the 990 and 995 class mainframes.

These parameters, and the VECTOR_FORTRAN command, are described in chapter 12.

Over-Subscripting of Arrays

FORTRAN Version 2 adheres more strictly to the language restriction on out-of-bounds subscripts of arrays than FORTRAN Version 1. This can lead to different answers in cases where out-of-bounds subscripts are used to access the same array element through different subscripts. Consider the following array:

(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)

FORTRAN stores arrays in column order. Therefore, element (1,2) immediately follows element (3,1) in memory. The following program segment works under FORTRAN Version 1 but is likely to fail under FORTRAN Version 2:

```
REAL A(3,4)
DATA I/4/
J=1
A(I,J) = 5.
A(2,3) = A(1,2)/A(2,2)
```

The compiler presumes that the reference to A(I,1) is accessing some element in column 1 of array A. It also knows that A(1,2) may well be done before the store into A(I,1) due to code scheduling. Of course, A(4,1) is in fact the location A(1,2) so the load picks up the wrong value. This occurs only when the compiler can determine a constant subscript value in the same dimension for the two references and constants differ.

In the above case, the constant value of J is inferred from the code leading up to the reference to A(I,J) so having all variable subscripts is not always sufficient to "trick" the optimizer. The best remedy is to avoid over-subscripting of arrays.

Intrinsic Functions

INSB and EXTB are expanded inline in FORTRAN Version 2 and out-of-line (called from the Math Library) in FORTRAN Version 1. In FORTRAN Version 1, an error with one of these intrinsic functions will be reported in your source program and can be handled using SYSTEMC. FORTRAN Version 2 programs will report an instruction specification error. You can force these out-of-line by adding an EXTERNAL statement for their name for each program unit where you want them out-of-line. FORTRAN Version 1 also permits INSB and EXTB to be used in compile time constant expressions. FORTRAN Version 2 does not initially allow this.

Uninitialized Variables

FORTRAN Version 2 keeps more items on the hardware stack than FORTRAN Version 1. This usually causes problems with programs that rely on uninitialized local variables being zero or on such variables retaining their value from one CALL to the next. This difference can be eliminated by adding a SAVE statement to the affected routine or by using the FORCED_SAVE parameter on the VECTOR_FORTRAN command.

Differences That Affect Debugging

Due to improved optimization in FORTRAN Version 2, the number of displayable variable values will diminish, particularly at OPTIMIZATION_LEVEL=HIGH and VECTORIZATION_LEVEL=HIGH. FORTRAN Version 2 assigns more variables to registers, and over larger regions of the program, than FORTRAN Version 1.

Line numbers at VECTORIZATION_LEVEL=HIGH bear little relation to the source because the vectorizer may distribute the code from one loop into several loops.

Stack Overflow During Execution of FORTRAN Programs

FORTRAN Version 2 uses the stack to hold temporary vector values. Expressions using large vectors may cause a stack overflow condition. This can be avoided by using the STACK_SIZE parameter of the EXECUTE_TASK command or by creating a program description for the program which contains a value for the stack size. The default stack size is around two million bytes. Increasing this to twenty million should allow most programs to execute. Note that stack overflow can also be a symptom of run-away recursion.

Differences Due to Vectorization

The following paragraphs describe differences between FORTRAN Version 1 and FORTRAN Version 2 that occur when the FORTRAN Version 2 program is compiled with VECTORIZATION_LEVEL=HIGH.

Adding Arrays of Numbers

The vectorizer uses the hardware instruction SUM when adding an array of numbers. The result of adding an array of numbers is dependent on the order in which they are added. The result of adding an array of numbers with vector instructions (all elements are added at once) may produce a result that is slightly different from that of the iterative DO loop due to roundoff errors. If you specify EXPRESSION_EVALUATION = MAINTAIN_PRECISION, the vectorizer will not use the SUM instruction.

Over-Subscripting of Arrays

Over-subscripting of arrays often causes the vectorizer to generate improper code. The model 990 and 995 mainframes have vector instructions that can process 512 array elements at a time. Therefore, if an array is larger than 512 elements, the vectorizer must place a loop around the vector instruction in order to process all elements; this is called stripmining. The decision to stripmine is based on the declared size of the array. For example:

```
      REAL A(1000)
      CALL SUB(A,1000)

      SUBROUTINE SUB(A,N)
      REAL A(100)
      DO 10 I=1,N
   10 A(I) = 0.0
```

The compiler will vectorize the DO loop, but it will not put a loop around the vector operation because the array is declared to have only 100 elements. In reality, the program is expecting 1000 elements to be zeroed out and this is exactly what would have happened under FORTRAN Version 1. In FORTRAN Version 2, with VECTORIZATION_LEVEL = HIGH, only 512 elements will be zeroed out.

One way to solve this problem is to declare A in the subroutine SUB as an assumed-size array, A(*), which means that it's length is not known. In the past, such arrays were traditionally declared as A(1). The compiler treats dummy argument arrays declared as A(1) as if they had been declared A(*). Therefore, such usages will not experience the problem although changing the dimension to A(*) is preferred.

A related problem can occur if a non-dummy argument array in a common block is over-subscripted (sometimes done to initialize the common block). This illegal practice has always been likely to fail, even under previous optimizers. Use equivalencing to obtain an array that spans the common block. For example:

```
    COMMON /BLK/A(100), B(100), C(100), D(400)
    DATA N/700/                                        Likely to produce errors
    DO 10 I=1,N
10  A(I) = 0
```

```
    COMMON /BLK/ A(100), B(100), C(100), D(400)
    REAL ALLBLK(700)
    EQUIVALENCE (ALLBLK(1), A(1))                      Correct
    DATA N/700/
    DO 10 I=1,N
10  ALLBLK(I) = 0
```

This problem also applies to over-addressing of blank common. To avoid the problem (which only occurs when the high level of vectorization is invoked), it is best to declare the over-indexed array with the maximum size it can take on, thus eliminating over-indexing. For example:

```
    COMMON // A(1)
       .
       .                    Likely to produce errors
       .
    DO 10 I=1,N
10  A(I) = 0
```

```
    COMMON // A(1000)
       .
       .                    Correct
       .
    DO 10 I=1,N
10  A(I) = 0
```

If this maximum size is unknown, then the array would be declared with a large size (greater than 512 elements).

Intrinsic Functions

The vectorizer vectorize references to a number of math library routines and uses vector versions of these routines. For example,

```
      REAL A(1000)
      DO 10 I=1,1000
   10 A(I) = SQRT(REAL(I))
```

is vectorized to use the vector version of SQRT. The result will be the same as the scalar version unless errors occur in computing the square root.

Under FORTRAN Version 1, a bad SQRT call would stop execution and sometimes, depending on OPTIMIZATION_LEVEL parameter specification, the value in I could be used to determine which element was bad.

Under FORTRAN Version 2, the vector routines will process 512 elements at a time and may encounter more than one error in the 512 elements. Therefore, you will get an error message about an invalid argument to SQRT but this may actually reflect multiple errors. This is only of concern if you were using SYSTEMC to intercept errors, correct them by supplying a valid result, and resuming execution. This will not work with the vector versions of these routines because multiple errors may occur and there is no easy way to tell which element generated the error. Such programs can be made to work by adding EXTERNAL statements for the affected math routines which will prevent the vectorizer from vectorizing them.

Execution Time

Very slightly faster code will be generated at OPTIMIZATION_LEVEL=HIGH and VECTORIZATION_LEVEL=HIGH if you specify DEBUG_AIDS=NONE on the VECTOR_FORTRAN command.

Index

A

A edit descriptor
  Character list items 7-31
  Noncharacter list items 7-33
ABORT_FILE parameter 12-42
ABORT subprogram 10-26
ABS intrinsic function 9-8
Absolute value 9-8
Absolute value, definition A-1
ACOS intrinsic function 9-8
Actual arguments 8-21
Adjustable arrays 3-2; 8-29
AIMAG intrinsic function 9-9
AINT intrinsic function 9-9
ALL intrinsic function 9-10
Allocatable array 3-2
ALLOCATE statement 3-5
ALLOCATED intrinsic function 9-10
Allocation of storage H-3
ALOG intrinsic function 9-11
ALOG10 intrinsic function 9-11
Alternate return 8-36
AMAX0 intrinsic function 9-11
AMAX1 intrinsic function 9-12
AMIN0 intrinsic function 9-12
AMIN1 intrinsic function 9-12
AMOD intrinsic function 9-13
AND intrinsic function 9-13
ANINT intrinsic function 9-14
ANSI extensions G-1
ANSI standard language, definition A-1
ANY intrinsic function 9-14
Apostrophe edit descriptor 7-27, 54
Application units 10-57
Arcsine 9-15
Arctangent 9-15
Argument
  Actual 8-21
  Array 8-26
  Association 8-23
  Asterisk 8-25
  Character 8-31
  Checking at load time 12-8
  Dummy 8-22
  Passing 8-20
  Subprogram names 8-25
  Variable 8-25
Arithmetic
  Assignment statement 5-25
  Boolean relational expressions 5-20
  Computation 5-1
  Constant expression 5-7
  Expression 5-2
  Format specification 7-25
  IF statement 6-5
  Operator precedence 5-6
  Operators 5-2
  Overflow 10-54
  Primary 5-2
ARITHMETIC_LOSS_OF_SIGNIFICANCE 10-54
Array
  Adjustable 3-2; 8-29
  Allocatable 3-2, 5
  Arguments 8-26
  Assignment 6-10
  Assumed-shape 3-2; 8-30
  Assumed-size 3-2; 8-29
  Bounds 3-1; 4-18
  Conformability 3-7
  Declaration 3-1; 4-18
  Dimensions 3-1
  Elements 3-11
  Expressions 5-24
  Fixed-size 3-2
  Interface block 4-30
  Intrinsic functions 9-4
  Lower bound 3-1
  Name 3-1
  Namelist 7-70
  Over-indexing 11-6; I-1, 2, 4
  Processing intrinsic functions 9-4
  References 3-10
  Runtime checks 12-18
  Section, definition A-1
  Sectioning 13-23
  Sections 3-14
  Shape 3-7
  Storage 3-3
  Type 3-1
  Upper bound 3-1
Array-valued expressions 5-1, 24
ASCII collating sequence 10-22, 48; C-2
ASIN intrinsic function 9-15
ASSIGN statement 6-4
Assigned GO TO statement 6-3
Assignment
  Arithmetic 5-25
  Boolean 5-29
  Character 5-27
  Logical 5-28
  Multiple 5-30
  Statements 5-25
  Syntax summary H-1
Association of arguments 8-23
Assumed-length character strings 8-32
Assumed-shape arrays 3-2; 8-30
Assumed-size arrays 3-2; 8-29
Asterisks as arguments 8-25
ATAN intrinsic function 9-15
ATANH intrinsic function 9-16
ATAN2 intrinsic function 9-16
Attach, definition A-1
Attributes of a file 7-3

Audience for this manual 7

B

BACKSPACE statement 7-109
Batch mode, definition A-1
Beginning of Information (BOI) 7-2, 108
Beginning of Information (BOI), definition A-1
Binary object code file 12-8
BINARY_OBJECT parameter 12-8
Binary object program, definition A-1
Binary zero fill 2-15
Bit, definition A-1
Blank common block, definition A-1
Blank common blocks 4-8
Blank fill 2-15
Block
  Data subprogram 8-2, 16
  IF statement 6-7
  IF structures 6-7
  WHERE statement 6-11
BLOCK DATA statement 8-16
BN edit descriptor 7-27, 52
BOI (see Beginning of Information)
BOOL intrinsic function 9-17
Boolean
  Assignment statement 5-29
  Constant 2-15
  Constant expression 5-18
  Conversion 9-17
  Data type 1-2; 2-15; 4-3
  Expression 5-17
  Format specification 7-25
  Operator 5-17
  Primary 5-17
  SCL value 10-15, 29, 33
  Variables 4-3
BOOLEAN statement 4-3
Broadcasting 13-21
BUFFER IN statement 7-72
BUFFER IN status 7-111
Buffer Input/Output 7-71
BUFFER OUT statement 7-73
BUFFER OUT status 7-111
Buffer statement, definition A-1
Byte
  Definition A-2
  Variables 4-4
BYTE statement 4-4
BZ edit descriptor 7-27, 52

C

C$ ASSUME directive 11-11; 13-29
C$ COLLATE directive 11-8
C$ CONTROL directive 11-12; 13-30
C$ directives
  Description 1-3; 2-5; 11-1
  Recognition 12-8
  Syntax 2-5
C$ DO loop 11-9
C$ ELSE directive 11-4
C$ ENDIF directive 11-4
C$ EXTEND directive 11-6
C$ EXTERNAL directive 11-10
C$ IF directive 11-4
C$ LIST directive 11-2
C$ PARAM directive 10-13; 11-13
C$ PP directive 11-15
C Routines D-7
C$ SEGFILE directive 11-7
CABS intrinsic function 9-17
Call by reference 9-2
Call by reference, definition A-2
Call by value 9-2
Call by value, definition A-2
CALL statement 8-9
Called program, definition A-2
Calling other language subprograms D-1
Calling program, definition A-2
Calling sequence, definition A-2
Carriage control 7-10
Catalog, definition A-2
Catalog Name, definition A-2
CCOS intrinsic function 9-18
CDC extensions 1-2; G-1
CEXP intrinsic function 9-18
CHAR intrinsic function 9-19
Character
  Arguments 8-31
  Assignment statements 5-27
  Constant expressions 5-13
  Constants 2-14
  Data
    Initialization 4-17
    Transfer 7-116
  Definition A-2
  Expressions 5-12
  Extended constant expressions 5-13
  Format specification 7-24
  Length 4-5
  Operator 5-12
  Primary 5-12
  Range 2-14
  Set 2-2
  Strings
    Assumed-length 8-32
    In expressions 5-7
  Substrings 4-6
  Variables 4-6
CHARACTER statement 4-5
CHGUCF call 10-54

CLOCK function 10-52
CLOG intrinsic function 9-19
Close operation, definition A-2
Close request, definition A-3
CLOSE statement 7-100
CLOSMS call 7-80
CMPLX intrinsic function 9-20
COBOL subprograms D-5
Collating sequence
  Control subprograms 10-48
  Definition A-3
  Description 10-48; 11-8
  Partial 10-48
  Selection 11-8; 12-9
Collating weight tables 11-8; C-1
Collation control 10-48
Collation control directive 11-8
Collation table, definition A-3
Collation tables
  Predefined 10-48; C-1
  Selection 11-8; 12-9
  User-defined C-1
Colon edit descriptor 7-58
COLSEQ call 10-49
COLSEQ call options C-1
Command Interface Subprogram (SCLCMD) 10-37
Comments
  Inline 2-4
  Line 2-4
Common and Equivalence map 12-32
Common Blocks
  Association 7-93
  Declaration 4-8
  Definition A-3
  Extensible 4-8; 11-6
  Referenced by program units 8-33
  Segment access file 4-8; 7-7; 11-6
  Size 4-8
Common logarithm 9-45
COMMON statement 4-8
Communicating
  Between procedures 8-9
  With the System Command Language 10-1
Compilation
  Command 12-1
  Conditional 11-4
  Directives 11-1; 12-8
  Errors 12-9, 24
  Statistics 12-24
  Time, definition A-3
COMPILATION_DIRECTIVES parameter 12-8
Compile command 12-1
Compile command parameters
  Formats 12-5
  Names 12-7
  Summary 12-3
Compiler
  Control directives 11-1
  Description 1-4
  Directive lines 2-5
  Errors 12-9
  Maps 12-26
  Optimization 12-16
  Output listing file 12-12, 24
  Processing 11-1
  Reference map 12-25
  Source listing 12-24
  Statistics 12-24
Compiling programs 12-1
COMPL intrinsic function 9-20
Complex
  Constant 2-20
  Conversion 9-20
  Variables 4-11
COMPLEX statement 4-11
Computed GO TO statement 6-2
Concatenation 5-12
Concatenation operator 5-12
Condition code 7-12; 10-9
Condition code, definition A-3
Condition name 10-4
Condition name, definition A-3
Condition numbers 7-12; 10-43
Conditional compilation directives 11-4
CONDNAM function 10-4
CONDSYM call 10-5
CONJG intrinsic function 9-21
Conjugate 9-21
CONNEC call 7-117
Connected files 7-120
Constant expressions 5-1
Constant integer expression 5-5
Constants
  Boolean 2-15
  Character 2-14
  Complex 2-20
  Double precision 2-21
  Extended Hollerith 2-18
  Hexadecimal 2-17
  Hollerith 2-15
  Integer 2-11
  Logical 2-22
  Octal 2-16
  Real 2-12
  Symbolic 4-35
Continuation lines 2-5
CONTINUE statement 6-21
Control Data Corporation Extensions 10; 1-2; 12-14, 16; G-1
Control transfer 8-36
Conventions 9
Conversion in arithmetic expressions 5-9
COS intrinsic function 9-21
COSD intrinsic function 9-22
COSH intrinsic function 9-22
Cosine 9-21
COTAN intrinsic function 9-23

Cotangent 9-33
COUNT intrinsic function 9-23
CPU seconds 10-52
CREATE_FILE_CONNECTION command 7-9
CREV call 10-27
CSIN intrinsic function 9-24
CSOWN call 10-51
CSQRT intrinsic function 9-24
CYBER Software Support Hotline 10
CYBIL procedures D-1

D

D edit descriptor 7-26, 34
DABS intrinsic function 9-24
DACOS intrinsic function 9-25
DASIN intrinsic function 9-25
Data
  Default type, definition A-3
  Format 7-22, 118
  Mismatches 7-118
  Type 4-1; 5-9, 18
DATA statement 4-12
DATA statement implied DO 4-15
DATAN intrinsic function 9-25
DATAN2 intrinsic function 9-26
DATE function 10-52
DBLE intrinsic function 9-26
DCOS intrinsic function 9-27
DCOSH intrinsic function 9-27
DDIM intrinsic function 9-27
DEALLOCATE statement 3-6
DEBUG_AIDS parameter 12-8
Debug utility commands file 12-42
Debug Utilty 1-6; E-1
Debugging
  Options 12-8; E-1
  Subprograms 10-41
Decimal digits 2-11
Declarations
  Array 3-1
  OPEN statement 7-95
  Summary H-2
Declaring arrays 3-1
DECODE statement 7-90
DEFAUA call 10-57
Default
  Data type, definition A-3
  Value, definition A-3
DEFAULT_COLLATION parameter 12-9
DELV call 10-28
Descriptor 7-26
DEXP intrinsic function 9-28
Diagnostic, definition A-3
Differences between FORTRAN 1 and FORTRAN 2 I-1
DIM intrinsic function 9-28
Dimension bounds 3-1, 11; 4-18
DIMENSION statement 4-18
DINT intrinsic function 9-29
Direct access
  Files
    Creating 7-84
    Overview 7-83
    Record number 7-83
  Input/Output, definition A-3
  READ statement 7-83
  Record length calculation 7-85
  WRITE statement 7-83
DISCON call 7-118
Disjunction, logical inclusive 5-17
DISPLA call 10-53
Display code, definition A-4
Divide fault 10-54
DLOG intrinsic function 9-29
DLOG10 intrinsic function 9-29
DMAX1 intrinsic function 9-30
DMIN1 intrinsic function 9-30
DMOD intrinsic function 9-30
DNINT intrinsic function 9-31
DO list, implied 4-15
DO loop
  Control directive 11-9
  Description 6-13
  Extended range 6-20
  In iolist 7-16
  Inner 6-16
  Interchange 13-24
  Iteration count 6-14
  Map 12-34
  Minimum trip count 6-13; 11-9; 12-15
  Nested 6-16
  Outer 6-16
  Range 6-13
  Variable 6-13
  Vectorization of 13-11
DO statement 6-13
Domain, definition A-4
DOTPRODUCT intrinsic function 9-31
Double precision
  Constant 2-21
  Variables 4-20
DOUBLE PRECISION statement 4-20
DPROD intrinsic function 9-32
DSIGN intrinsic function 9-32
DSIN intrinsic function 9-32
DSINH intrinsic function 9-33
DSQRT intrinsic function 9-33
DTAN intrinsic function 9-33
DTANH intrinsic function 9-34
Dummy arguments 8-22
DUMP call 10-41

E

E edit descriptor 7-26, 35
EBCDIC, definition A-4
Edit descriptor
  Boolean 7-26
  Character 7-26
  End of record 7-27
  Logical 7-26
  Nonrepeatable 7-27, 50
  Numeric 7-26
  Repeatable 7-26, 29
  Scale factor 7-27
  Tabulation control 7-16
Elemental intrinsic functions 9-4
Elements in an array 3-11
ELSE IF statement 6-7
ELSE statement 6-7
ELSEWHERE statement 6-11
ENCODE statement 7-90
END IF statement 6-7
END INTERFACE statement 4-30
End-of-File (EOF) 7-112
End-of-File (EOF), definition A-4
End-of-Information (EOI) 7-2
End-of-Partition (EOP), definition A-4
End-of-Record edit descriptor 7-58
END specifier 7-12
END statement 6-24
ENDFILE statement 7-110
ENDWHERE statement 6-11
ENTER_PPE command F-5
ENTER_PROGRAMMING_ENVIRONMENT command F-1
ENTPE command F-1
Entry point
  Definition A-4
  In programs 8-34
  Map 12-35
  Name 8-5
ENTRY statement 8-34
Environment
  NOS/VE 1-6
  Professional programming F-5
  Programming F-1
EOF (see End-of-File)
EOI (see End of Information)
Equivalence
  And common 4-21
  Class 4-21; A-4
  Function 9-34
  Logical 5-14
EQUIVALENCE statement 4-21
EQV intrinsic function 9-34
ERF intrinsic function 9-35
ERFC intrinsic function 9-35
ERR specifier 7-12
ERROR_LEVEL parameter 12-10
ERROR parameter 12-9
Errors
  Compilation 12-9, 24
  Execution 10-42
  Function 9-35
  Handling subprograms 10-42
  In this manual 10
  Numbers 7-12; 10-43
  Runtime 10-43
  Severity level 12-10
Evaluation
  Arithmetic expressions 5-9
  Expressions 5-22
Examples
  Calling other languages D-1
  Mass storage 14-10
  Program 14-1
Executing programs 12-1, 40
Execution
  Command
    Format 12-40
    Parameter example 14-12
    Parameter subprograms 10-14
    Parameters 10-12, 14; 11-13; 12-43
  Parameter definition 10-12
  Termination 10-26
  Time
    Definition A-4
    Differences from Version 1 I-6
    Format specification 7-25
    Improving, see For Better Performance
EXIT call 10-54
EXP intrinsic function 9-36
Explicit vectorization 13-6
EXPONENT_OVERFLOW 10-54
EXPONENT_UNDERFLOW 10-54
Exponential function 9-36
Exponentiation 5-6, 9
EXPRESSION_EVALUATION parameter 12-10
Expression, extended constant 5-1
Expressions
  Arithmetic 5-2
  Arithmetic and boolean relational 5-20
  Arithmetic constant 5-7
  Arithmetic constant extended 5-8
  Arithmetic interpretation 5-9
  Array-valued 5-1, 24
  Boolean 5-17
  Boolean constant 5-18
  Boolean constant extended 5-18
  Character 5-12
  Character constant 5-13
  Character constant extended 5-13
  Character relational 5-21
  Constant 5-1
  Dimension bound 3-1
  Evaluation 5-9, 22; 12-10
  Extended integer constant 5-8
  Integer 5-4

Integer constant 5-8
Logical 5-14
Logical constant 5-16
Logical constant extended 5-16
Relational 5-19
Rules 5-22
Scalar 5-1
EXTB intrinsic function 9-36
Extended
Arithmetic constant expression 5-8
Boolean constant expression 5-18
Constant expression 5-1
Hollerith constant 2-18
Integer constant expression 5-8
Internal files 7-89
Logical constant expression 5-16
Range DO loops 6-20
Extensible common blocks 4-8; 11-6
Extensible common, definition A-4
Extensions to ANSI FORTRAN 1-2
External
Control 11-10
File, definition A-4
Function 4-24; 8-12
Function reference 8-14
Procedures 11-10
Reference, definition A-4
Storage device, definition A-5
Subroutine 4-24
EXTERNAL statement 4-24
Extract bits function (EXTB) 9-36

F

F edit descriptor 7-26, 39
F record type, definition A-5
Fast I/O 7-119
Field, definition A-5
File
Association with unit number 7-95
Attributes
Changing 7-5
Default 7-3
Attributes, definition A-5
Closing 7-100
Compiler output listing 12-12
Connected 7-119, 120
Connection routines 7-117
Creating 7-95
Cycle, definition A-5
Definition A-5
Direct access 7-83
Disassociation with a unit number 7-101
Extended internal 7-89
Information 7-102
Input source code 12-12
Internal 7-86
Job log 10-53
Keyed 7-6
Name substitution 12-43
Open sharing 7-119
Opening 7-7, 95
Overview 7-2
Physical 7-9
Position 7-108
Position, definition A-5
Positioning statements 7-108
Random 7-74
Random access 7-6
Reference, definition A-5
References 7-2
SCL value 10-16
Segment access 7-7, 93
Sequential access 7-6
Standard internal 7-86
Standard system 7-9
Substitution 12-43
Unit 7-8
File Management Utility (FMU) 1-6
File Migration Aid (FMA) 1-6
Fixed-size array 3-2
FLOAT intrinsic function 9-37
Floating point number, definition A-6
Flow control statements 6-1
Flow control syntax summary H-5
FLP$FORMAT_MESSAGE call 10-6
FMT specifier 7-11, 22
For Better Performance 2-21; 3-3, 6, 13; 4-10, 20, 29; 6-15; 7-9, 75, 94; 8-11, 13, 32; 12-15, 16, 17, 20, 41; 13-12
FORCED_SAVE parameter 12-11
Format specification
Arithmetic or boolean 7-25
Character 7-24
Description 7-22
FMT 7-11
Format specifier 7-11
FORMAT statement 7-23
Formatted
Input/Output 7-18
PRINT statement 7-21
PUNCH statement 7-21
READ statement 7-18
WRITE statement 7-20
Formatting of data 7-22
FORTRAN
Callable subprograms 10-1
Character set 2-2
Collating sequence control subprograms 10-48
Compiler 1-4; 12-1
Debugging subprograms 10-41
Default file attributes 7-4
Error handling subprograms 10-42
Example programs 14-1
Execution command 12-40
Fast Input/Output 7-119
Language Summary H-1
Main program 8-2
Program overview 8-2

Statement 2-4
Statement functions 8-2
Subprograms 8-7
System Command Language subprograms 10-1
Utility subprograms 10-1
FORTRAN Version 2 language 1-1, 4
FORTRAN Version 2 language differences I-1
FORTRAN2 (See VECTOR_FORTRAN command)
FP_INDEFINITE 10-54
FP_LOSS_OF_SIGNIGICANCE 10-54
Free form
Input 7-61
Output 7-64
FTN2 (See VECTOR_FORTRAN Command)
Function
Executing 8-14, 18
External 8-12
Intrinsic 9-1
Reference 8-14
Statement 8-17
Subprogram 8-12
FUNCTION statement 8-12

G

G edit descriptor 7-26, 41
Gathering 13-17
General
Subprograms 1-7; 10-1
Utilities 1-7; 10-1
Generic function name, definition A-6
Generic name of intrinsic functions 9-3
GETBVAL call 10-15
GETCVAL call 10-16
GETIVAL call 10-17
GETSCNT function 10-18
GETSVAL call 10-19
GETVCNT function 10-20
GETVREF call 10-21
Glossary A-1
GO TO statements
Assigned 6-3
Computed 6-2
Unconditional 6-1
Graphic character, definition A-6

H

H edit descriptor 7-27, 53
Hexadecimal constant 2-17
Hollerith constant 2-15
Hotline, CYBER Software Support 10
Hyperbolic
Cosine 9-27
Sine 9-33, 62
Tangent 9-34, 66

I

I edit descriptor 7-26, 42
IABS intrinsic function 9-37
ICHAR intrinsic function 9-37
Identification information 2-6
IDIM intrinsic function 9-38
IDINT intrinsic function 9-38
IDNINT intrinsic function 9-38
IF statement
Arithmetic 6-5
Block 6-7
Logical 6-6
Nested 6-9
IFIX intrinsic function 9-39
IGNFDM call 7-118
IM/DM commands 1-7
IMPLICIT NONE statement 4-26
IMPLICIT statement 4-26
Implicit type 4-26
Implicit type, definition A-6
IMPLICIT type statement 4-26
Implicit vectorization
C$ directives 13-29
Overview 13-9
Report 13-31
Implied DO list 4-15; 7-15
Indefinite value
Definition A-6
Testing for 10-42
INDEX intrinsic function 9-39
Index key types in a random file 7-75
Index of a random file 7-75
Induction variable elimination 13-25
Infinite value
Definition A-6
Testing for 10-42
Initial lines 2-5
Initializing values 4-12; 8-16
Inline comments 2-4
Inline intrinsic functions 9-1
Inline intrinsic functions, definition A-6
Inner DO loop 6-16
Input
List directed 7-61
Source code file 12-12
$INPUT file 7-9
Input/Output
Buffer 7-71
Character data format specification 7-24
Direct access 7-6
Edit descriptors 7-26
Execution time format specification 7-25
Fast 7-119
Format specification 7-22
Formatted 7-18
Internal 7-86
Internal data transfer 7-115
Introduction 7-1

- List 7-13
- List directed 7-61
- Lists 7-13
- Mass storage 7-74
- Namelist 7-65
- Random 7-6
- Related statements 7-95
- Sequential 7-6
- Statement specifiers 7-11
- Statements summary H-7
- Status checking routines 7-111
- Unformatted 7-59
- Units 7-8
- Units map 12-38

INPUT parameter 12-12
Input source map 12-12
INPUT_SOURCE_MAP parameter 12-12
Inquire
- By file 7-102
- By unit 7-102

INQUIRE statement 7-102
INSB intrinsic function 9-40
Insert bits function (INSB) 9-40
Instance of open, definition A-6
INT intrinsic function 9-41
INTCOND function 10-9
Integer
- Constant expression 5-7
- Constants 2-11
- Expressions 5-4
- Range 2-11
- SCL value 10-17, 31
- Variables 4-29

INTEGER statement 4-28
Interactive mode, definition A-6
INTERFACE statement 4-30
Internal
- Data transfer routines 7-115
- File 7-86
- File, definition A-6
- Input/Output 7-86
- READ statement 7-88
- WRITE statement 7-87

Intrinsic declaration 4-32
Intrinsic function promotion 13-22
Intrinsic functions
- Array-processing 9-4
- Description 9-1
- Differences from Version 1 I-2, 6
- Domain 9-6
- Elemental 9-4
- Generic name 9-3
- Inline 9-1
- Mathematical 9-6
- Range 9-6
- Specific name 9-3
- User-supplied 4-32

INTRINSIC statement 4-32
IOCHEC function 7-113
IOSTAT specifier 7-12
ISIGN intrinsic function 9-41
Iteration count 6-13

J

JDATE function 10-52
Job, definition A-7
Job log
- Definition A-7
- File 10-53

Julian date function 10-52

K

Key, SCL value 10-16
Keyed-file
- Calls 1-7
- Organizations A-7

Keyword, definition A-7

L

L edit descriptor 7-44
Label, statement 2-5; 6-4
Language
- Elements 2-1
- Summary H-1

LBOUND intrinsic function 9-42
LEGVAR function 10-42
LEN intrinsic function 9-42
LENGTH function 7-113
LENGTHB function 7-114
LENGTHX subroutine 7-113
LGE intrinsic function 9-43
LGO file 12-40
LGT intrinsic function 9-43
Library, definition A-7
LIMERR call 10-47
Lines
- Comment 2-4
- Compiler directive 2-5
- Continuation 2-5
- Initial 2-5
- Program 2-4

List directed
- Input 7-61
- Output 7-64
- PRINT statement 7-64
- PUNCH statement 7-64
- READ statement 7-61
- WRITE statement 7-64

LIST_OPTIONS parameter 12-13
LIST parameter 12-12
Listing control directives 11-2
LLE intrinsic function 9-44
LLT intrinsic function 9-44
Load time, definition A-7
Loader control 11-6
Local file, definition A-7
Local path, definition A-7

LOG intrinsic function 9-45
Logarithm base 10 9-45
Logarithm intrinsic function 9-45
Logical
  Assignment statement 5-28
  Conjunction 5-14
  Constant 2-22
  Constant expression 5-16
  Equivalence 5-14
  Exclusive disjunction 5-14
  Expression 5-14
  IF statement 6-6
  Negation 5-14
  Nonequivalence 5-14
  Operation 5-14
  Operator 5-14
  Primary 5-14
  Values 2-22
  Variables 4-34
  WHERE statement 6-10
LOGICAL statement 4-34
Login, definition A-7
Logout, definition A-7
LOG10 intrinsic function 9-45
Loop collapse 13-28
Loop interchange 13-24
Lower bound of an array 3-1

M

MACHINE_DEPENDENT
 parameter 12-14
Main program 8-4
Manual
  Errors 10
  Online B-1
  Ordering B-1
  Organization 8
Maps
  Common and equivalence 12-32
  DO loops 12-34
  Entry points 12-35
  General format 12-26
  Input/output units 12-38
  Namelist 12-31
  Procedures 12-36
  Reference 12-25
  Statement labels 12-33
  Symbolic constants 12-30
  Unclassified name 12-39
  Variables 12-28
MASK intrinsic function 9-46
Mass storage
  Closing 7-80
  CLOSMS statement 7-80
  Definition A-7
  Examples 14-10
  File, definition A-7
  Input/Output 7-74
  Input/Output, definition A-8
  Opening 7-77
  OPENMS call 7-77
  READMS statement 7-80
  Selecting an index 7-81
  STINDX call 7-81
  Transmitting from 7-80
  WRITMS statement 7-78
Master index in random files 7-75
Math Library 9-1
Math Library, definition A-8
Math Library errors 10-44
Mathematical intrinsic functions 9-6
MATMUL intrinsic function 9-47
MAX intrinsic function 9-48
MAXVAL intrinsic function 9-48
MAX0 intrinsic function 9-49
MAX1 intrinsic function 9-49
Memory dump 10-41
MERGE intrinsic function 9-50
MIN intrinsic function 9-50
Minimum trip count 6-13; 11-9; 12-15
MINVAL intrinsic function 9-51
MIN0 intrinsic function 9-51
MIN1 intrinsic function 9-52
MOD intrinsic function 9-52
Module, definition A-8
Modulus 9-52
MOVLCH call 7-116
MOVLEV call 7-115
MSIO (see Mass storage input/output)
Multilevel file indexing 7-76
Multiple
  Assignment statement 5-30; H-1
  Entry 8-34
  Entry points 8-34

N

Name index key 7-75
Name, SCL value 10-16
Named common block
  Association with segment access
   files 11-7
  Creation 4-8
  Definition A-8
  Extensible 11-6
  Segment access files 7-93
Namelist
  Arrays 7-70
  Group 7-65
  Input 7-66
  Input/Output 7-65
  Map 12-31
  Output 7-69
  PRINT statement 7-69
  PUNCH statement 7-69
  READ statement 7-66
  WRITE statement 7-69
NAMELIST statement 7-65
Natural logarithm 9-45

Negation, logical 5-14
NEQV intrinsic function 9-52
Nested
  Block if structures 6-9
  DO loops 6-16
  Implied DO lists 7-16
NINT intrinsic function 9-53
Non ANSI features 1-2
Nonequivalence function 9-53
Nonequivalence, logical 5-14
Nonrepeatable edit descriptors 7-27, 50
Normalized floating point number, definition A-8
NOS/VE
  Communicating with 10-11
  Definition A-8
  Environment 1-6
  FORTRAN 1-1
NOS/VE commands, executing 10-27
NOS/VE status subprograms 10-2
NOS/VE subprograms 10-1
$NULL file 7-9
NULL file 7-9
Number index key 7-75
NUMERR utility subprogram 10-47

O

O edit descriptor 7-26, 45
Object code
  Definition A-8
  Execution 12-40
Object Code Utilities 1-6
Object library, definition A-8
Object program name 8-5
OCM 1-6
Octal
  Constant 2-16
  Format 7-45
ONE_TRIP_DO parameter 12-15
Online manuals B-1
Open, definition A-8
Open sharing 7-119
OPEN statement 7-95
Opening files 7-7, 94
OPENMS call 7-77
Operator precedence
  Arithmetic 5-6, 22
  Character 5-12
  Logical 5-16, 22
Operators
  Arithmetic 5-2
  Boolean 5-17
  Characters 5-12
  Concatenation 5-12
  Logical 5-14
  Relational 5-19
Optimization, definition A-8
OPTIMIZATION_LEVEL parameter 12-16
OPTIMIZATION_OPTIONS parameter 12-17
OR intrinsic function 9-54
OSV$ASCII6_FOLDED collating sequence table C-6
OSV$ASCII6_STRICT collating sequence table C-8
OSV$COBOL6_FOLDED collating sequence table C-10
OSV$COBOL6_STRICT collating sequence table C-12
OSV$DISPLAY63_FOLDED collating sequence table C-14
OSV$DISPLAY63_STRICT collating sequence table C-16
OSV$DISPLAY64_FOLDED collating sequence table C-18
OSV$DISPLAY64_STRICT collating sequence table C-20
OSV$EBCDIC collating sequence table C-22
OSV$EBCDIC6_FOLDED collating sequence table C-29
OSV$EBCDIC6_STRICT collating sequence table C-31
Outer DO loop 6-16
Output
  Free form 7-64
  List directed 7-64
  Listing 12-12
$OUTPUT file 7-9
OUTPUT file 7-9
Over-indexing of arrays 11-6; I-2, 4
Oversubscripting of arrays 11-6; I-2, 4

P

P edit descriptor 7-27, 50
PACK intrinsic function 9-54
Page eject 7-10
Parameter, definition A-9
PARAMETER statement 4-35
Parity errors 7-113
Partition boundary 7-2
Pascal triangle 14-1
Pass by reference, definition A-9
Pass by value, definition A-9
Passing
  Arguments 8-9
  Arrays 8-26, 31
  Character data 8-31
  Subprogram names 8-25
  Variables 8-25
Path, definition A-9
PAUSE statement 6-22
PDUMP utility subprogram 10-41
Performance hints (See For Better Performance)
Permanent file, definition A-9
Physical file 7-9

Positions 73 and beyond 2-6
Precedence among operators 5-22
Preprocessor control directives 11-15
Principal square root function 9-63
$PRINT_LIMIT parameter 12-41
PRINT statement
  Formatted 7-21
  List directed 7-64
  Namelist 7-69
Printer control 7-10
Printer output record 7-10
Procedure
  Argument association 8-23
  Communication 8-9
  Definition A-9
  Entry 8-34
  Maps 12-36
  Return 8-36
Procedure, definition A-9
PRODUCT intrinsic function 9-55
Professional Programming Environment 1-6; F-5
Program
  Examples 14-1
  Name 8-5
  Termination 10-54
  Unit
    Definition A-9
    Description 8-1
Program-library list, definition A-9
PROGRAM statement
  File substitution 12-43
  Format 8-5
Programming Environment 1-3; F-1
Programming Environment, definition A-9
PTR intrinsic function 9-56; D-8
PUNCH file 7-9
PUNCH statement
  Formatted 7-21
  List directed 7-64
  Namelist 7-69

Q

Quote edit descriptor 7-27, 54

R

R edit descriptor 7-26, 47
Random access
  Definition A-9
  Files 7-6
Random file index 7-74
Random files 7-74
  Index keys 7-74
  Master index 7-75
  Multilevel file indexing 7-76
  Subindex 7-75
Random number generation 9-56; 10-40
RANF intrinsic function 9-56; 10-40
Range, definition A-9
RANGET call 10-40
RANK intrinsic function 9-57
RANSET call 10-40
READ statement
  Direct access 7-83
  Formatted 7-18
  Internal 7-88
  List directed 7-61
  Mass storage 7-80
  Namelist 7-66
  Unformatted 7-60
READMS call 7-80
Real
  Constants 2-12
  Range 2-13
  Variables 4-37
REAL
  Intrinsic function 9-57
  Statement 4-37
REC specifier 7-12
Record
  Bytes 7-114
  Definition A-9
  Key 7-75
  Length
    Calculation in direct access files 7-86
    Definition A-10
  Number 7-83
Records 7-1
Recurrence cycles 13-13
REDBVAR call 10-29
REDCVAR call 10-30
REDIVAR call 10-31
REDSVAR call 10-32
Reference listing, definition A-10
Reference map 12-25
Related manuals B-1
Relational
  Character expressions 5-21
  Expression 5-19
  Operators 5-19
Relocatable, definition A-10
REMARK call 10-53
Repeatable edit descriptors 7-26, 29
Report messages 13-35
REPORT_OPTIONS parameter 12-18
RETURN statement 8-36
Rewind, definition A-10
REWIND statement 7-108
Runtime
  Definition A-10
  Error messages 10-43
  Format specification 7-25
  Parameters 10-13
RUNTIME_CHECKS parameter 12-18

S

S, SP, SS edit descriptors 7-27, 52
Sample Debug session E-1
SAVE statement 4-38
Scalar
  Definition A-10
  Expansion 13-20
  Expression 5-1
  Operations 13-4
  Values 2-1
Scale factor edit descriptor 7-27
Scattering 13-16
SCL parameters on the execution command 12-43
SCL (see System Command Language)
SCLCMD call 10-37
SCLKIND call 10-22
SCU 1-6
SECOND utility subprogram 10-52
Seed of RANF 10-40
Segment access file
  Common blocks 11-7
  Definition A-10
  Description 7-7, 93; 11-7
SEQ intrinsic function 9-58
Sequence vector 13-18
Sequential access
  Definition A-10
  Files 7-6
Set bits function 9-65
SHAPE intrinsic function 9-59
Sharing storage 4-21
SHIFT intrinsic function 9-60
Sign, definition A-10
SIGN intrinsic function 9-60
SIN intrinsic function 9-61
SIND intrinsic function 9-61
Sine intrinsic function 9-24, 32
SINH intrinsic function 9-62
SIZE intrinsic function 9-62
Slash edit descriptor 7-58
SNGL intrinsic function 9-63
Sort/Merge Utility 1-3
Source code, definition A-10
Source Code Utility 1-6
Source library, definition A-10
Source listing
  Creation 12-12
  Definition A-11
  Description 12-24
Specific function name, definition A-11
Specific name of intrinsic functions 9-3
Specification statements 4-1
Specifier
  END 7-12
  ERR 7-12
  FMT 7-11, 22
  Format 7-11
  IOSTAT 7-12
  REC 7-12
  UNIT 7-11
SQRT intrinsic function 9-63
Square root functions 9-63
Square root intrinsic function 9-24
SSWITCH utility subprograms 10-53
Stack overflow I-3
Standard
  Features 1-1
  System file 7-9
  Units 7-9
STANDARDS_DIAGNOSTICS parameter 12-19
Statement
  Function 8-2, 17
  Labels 2-5; 6-4
  Labels map 12-33
  Length 2-4
  Specification 4-1
  Specifiers for I/O 7-11
  Syntax summary H-1
  Writing 2-2
Status checking routines 7-111
Status condition code, definition A-11
STATUS parameter 12-19, 42
Status variable 10-19, 32, 36
Status variable, definition A-11
STINDX call 7-81
STOP statement 6-23
Storage allocation syntax summary H-2
STRACE call 10-41
String, definition A-11
String, SCL value 10-16, 30, 35
Stripmining 13-26
Subexpressions 5-2
Subindex in random files 7-75
Submitting comments 10
Subprograms
  Arguments 8-25
  Block data 8-16
  Collating sequence control 10-48
  Debugging 10-41
  Error handling 10-42
  External 11-10
  FORTRAN callable 10-1
  Function 8-12
  Other languages D-1
  Overview 8-7
  Parameter interface 10-14
  SCL variable 10-25
  Subroutines 8-8
  Utility 1-7; 10-38
Subroutine
  Call 8-9
  Description 8-2, 8
  Executing 8-9
  Return 8-36
  Subprograms 8-2, 8
SUBROUTINE statement 8-8
Subscript expression 3-1, 11
Substring, definition A-11
Substrings 4-6

Substrings in arrays 3-17
SUM intrinsic function 9-64
SUM operation 13-19
Sum reduction 13-19
SUM1S intrinsic function 9-65
Symbolic constants 4-35
Symbolic constants map 12-30
Symbolic names 2-9
System clock function 10-52
System Command Language (SCL)
  Command string 10-37
  Definition A-11
  Description 1-3
  STATUS parameter 12-42
  Status variable 10-2, 19, 32, 36; 12-19.
  Variables 10-25
System conditions 10-54
System sense switches 10-53
SYSTEM utility subprogram 10-44
SYSTEMC utility subprogram 10-44

T

T, Tl, TR edit descriptors 7-27, 56
Tabulation control 7-27
TAN intrinsic function 9-65
TAND intrinsic function 9-66
Tangent 9-65
TANH intrinsic function 9-66
TARGET_MAINFRAME parameter 12-20
Task, definition A-11
Task time 10-52
TERMINATION_ERROR_LEVEL parameter 12-21
TIME function 10-52
Total compilation time 12-24
Traceback
  Definition A-11
  Information 10-41
Traceback file 12-42
Transfers within a DO loop 6-16
Transmitting data items 7-13
Truncation intrinsic function 9-29
Truth table 5-15
TSTPARM function 10-23
TSTRANG function 10-24
Type conversion
  Boolean 9-17
  Complex 9-20
  Double precision 9-26
  Real 9-57
Type of arithmetic expression 5-9
Type statements
  Boolean 4-3
  Byte 4-4
  Character 4-5
  Complex 4-11
  Double precision 4-20
  Integer 4-28
  Logical 4-34
  Real 4-37
  Syntax summary H-1
Typing
  Explicit 4-3
  Implicit 4-26

U

U record type, definition A-11
UBOUND intrinsic function 9-67
Unclassified names map 12-39
Unconditional GO TO statement 6-1
Unformatted
  Input/Output 7-59
  READ statement 7-60
  WRITE statement 7-59
Unit association 7-9
Unit identifier, definition A-11
UNIT routine 7-111
UNIT specifier 7-11
Unlabeled common blocks 4-8
UNPACK intrinsic function 9-68
UPKSTAT call 10-10
Upper bound of an array 3-1
User name, definition A-11
User path, definition A-12
Using Debug E-2
Utility calls
Utility, definition A-12
Utility subprograms 1-7; 10-1, 38

V

V record type, definition A-12
Variable argument 8-25
Variable interface subprograms 10-25
Variable-length integer expressions 5-4
Variable-length logical expression 5-15
Variables
  Boolean 4-3
  Character 4-6
  Complex 4-11
  Declaring 2-23
  Double precision 4-20
  Integer 4-29
  Logical 4-34
  Map 12-28
  Real 4-37
  Type of 2-23
VECF (see VECTOR_FORTRAN command)
Vector 13-1
Vector, definition A-12
VECTOR_FORTRAN command
  Default values 12-3
  Description 12-1
  Examples 12-22
  Format 12-1

Parameter
Names 12-7
Short form 12-3
Table 12-3
Vector operations 13-15
Vectorization
Control 11-11
Control directives 13-29
Definition A-12
Explicit 13-6
Implicit 13-9
Messages 13-35
Overview 13-1
Report 12-18; 13-31
VECTORIZATION_LEVEL parameter 12-21
VFORTRAN (see VECTOR_FORTRAN command)
VFTN (see VECTOR_FORTRAN command)

W

Weight tables 10-48
WHERE statement
Block 6-11
Logical 6-10
Word alignment 4-21
Word, definition A-12
Working catalog, defintion A-12
Working storage area, definition A-12
WRITE statement
Direct access 7-85
Formatted 7-20
Internal 7-87
List directed 7-64
Mass storage 7-78
Namelist 7-69
Unformatted 7-59
Writing FORTRAN statements 2-2
WRITMS call 7-78
WRTBVAR call 10-33
WRTCVAR call 10-34
WRTIVAR call 10-35
WRTSVAR call 10-36
WTSET call 10-50

X

X edit descriptor 7-27, 54
XOR intrinsic function 9-69

Z

Z edit descriptor 7-26, 48

Comments (continued from other side)

Please fold on dotted line;
seal edges with tape only.

FOLD

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA
Technical Publications
SVL104
P.O. Box 3492
Sunnyvale, CA 94088-3492

FORTRAN Version 2 for NOS/VE Language Definition Usage 60487113 D

We would like your comments on this manual to help us improve it. Please take a few minutes to fill out this form.

Who are you?

- ☐ Manager
- ☐ Systems analyst or programmer
- ☐ Applications programmer
- ☐ Operator
- ☐ Other ____________

How do you use this manual?

- ☐ As an overview
- ☐ To learn the product or system
- ☐ For comprehensive reference
- ☐ For quick look-up
- ☐ Other ____________

What programming languages do you use? ____________

How do you like this manual? Answer the questions that apply.

Yes	Somewhat	No	
☐	☐	☐	Does it tell you what you need to know about the topic?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple? ☐ Too complex?)
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Do you use this manual frequently?

Comments? If applicable, note page and paragraph. Use other side if needed.

Check here if you want a reply: ☐

Name ____________ Company ____________

Address ____________ Date ____________

Phone ____________

Please send program listing and output if applicable to your comment.

CONTROL DATA